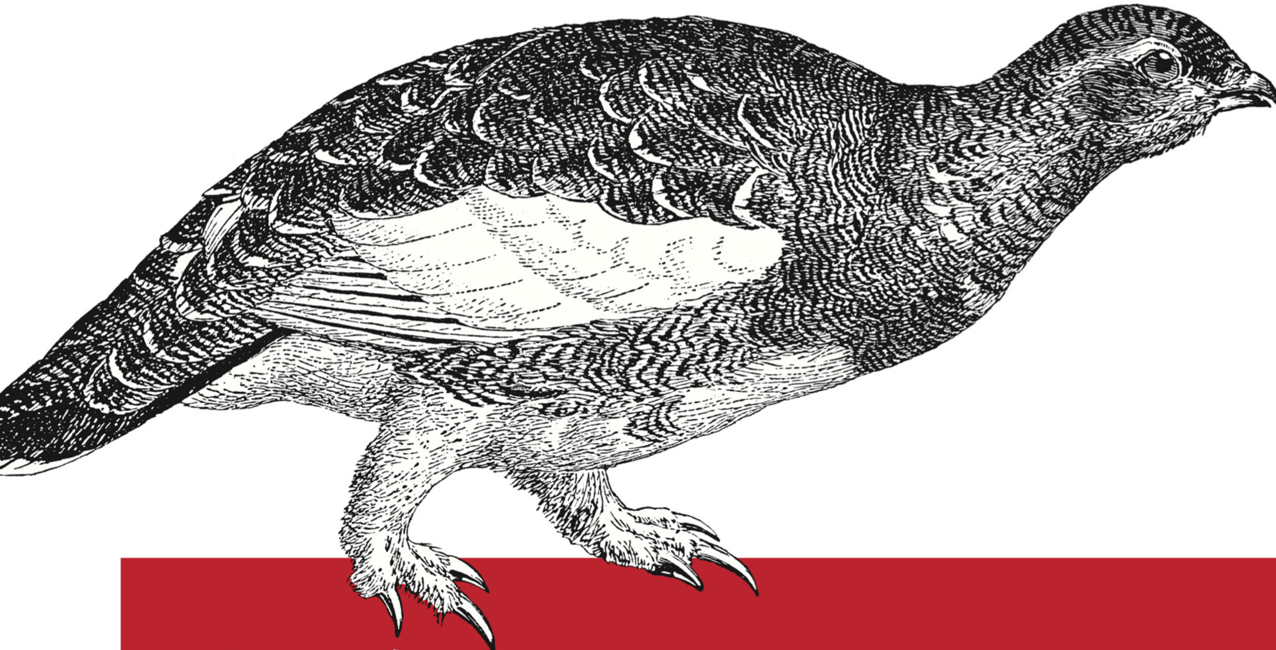
O'REILLY®

TURING 图灵程序设计丛书



数据科学入门

Data Science from Scratch

[美] Joel Grus 著

高蓉 韩波 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

高蓉

讲师，任教于杭州电子科技大学经济学院金融系。博士和硕士毕业于南开大学经济学院金融系，本科毕业于南开大学数学学院计算数学专业。研究领域包括资产定价、金融工程、计量经济应用、数据科学应用。

韩波

自由译者、撰稿人，从事IT工作二十余年，主要兴趣领域为机器学习、Python。曾为多家IT媒体撰稿，另有译著《Python数据分析》。

TURING 图灵程序设计丛书

数据科学入门

Data Science from Scratch

First Principles with Python

[美] Joel Grus 著
高蓉　韩波 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（C I P）数据

数据科学入门 / （美）格鲁斯（Grus,J.）著 ；高蓉，
韩波译. -- 北京 ：人民邮电出版社，2016.3
（图灵程序设计丛书）
ISBN 978-7-115-41741-1

Ⅰ. ①数… Ⅱ. ①格… ②高… ③韩… Ⅲ. ①数据处
理 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2016)第025033号

内 容 提 要

本书基于易于理解且具有数据科学相关的丰富的库的 Python 语言环境，从零开始讲解数据科学工作。具体内容包括：Python 速成，可视化数据，线性代数，统计，概率，假设与推断，梯度下降法，如何获取数据，*k* 近邻法，朴素贝叶斯算法，等等。作者借助大量具体例子以及数据挖掘、统计学、机器学习等领域的重要概念，详细展示了什么是数据科学。

本书适合有志成为数据科学工作者以及想了解数据科学的读者阅读。

◆ 著　　　　[美] Joel Grus
译　　　　高　蓉　韩　波
责任编辑　朱　巍
执行编辑　张　曼
责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：19
字数：451千字　　　　2016年 3 月第 1 版
印数：1－3 500册　　　2016年 3 月北京第 1次印刷
著作权合同登记号　图字：01-2015-8112号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

数据科学

有人称数据科学家为“21 世纪头号性感职业”（https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century/）。虽说如此称呼有些夸张，但这个名称对数据科学的推崇却一点也没错，这是一个蓬勃发展、前途无限的行业。很多分析师都预言，未来十年会需要比现在多得多的数据科学工作者。

那么，什么是数据科学？唯有正确理解数据科学，才能培养出数据科学家。根据广受业界赞誉的文氏图（http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram），数据科学是以下几个方面的交叉：

- 黑客技能
- 数学和统计学知识
- 专业技能

我原本很想写一本能涵盖以上三个方面的书，但很快意识到仅关于专业技能的撰写就会耗费上万页笔墨，于是及时放弃转而专注于前两个方面。我的目标有两个：一是帮助读者掌握从事数据科学工作所必需的黑客技能；二是帮助读者熟悉数学和统计学，这是数据科学的核心。

对一本书来说，这两个愿望有点大了。学习黑客技能的最好方法就是钻研技术。通过阅读本书，你可以理解我钻研技术的方式，但相同的方式对你未必最适合；你可以理解我使用的一些工具，但相同的工具对你来说未必最顺手；你可以理解我如何解决数据问题，但相同的方式对你来说未必最有效。举例的目的和希望是启发你以自己的方式和方法完成工作。本书涵盖的所有代码和数据都可以从 GitHub 上下载。

同样，学习数学的最好方式就是研习数学。当然本书并不是一部数学著作，我们在本书中大半也不会“研习数学”，我想强调的是数学知识对从事数据科学工作至关重要。不理解

概率、统计、线性代数，就无法真正开始数据科学工作。在需要的地方，书中会引入数学方程式、数学直觉、数学公理，以及借以阐释大数学思想的卡通漫画。有我在，别怕！

总之，数据科学相当有趣（尤其和税务筹划或者煤矿开采等其他工作相比）。

从零开始

很多很多的数据科学库、框架、模块、工具箱可以有效地实现数据科学大部分常见的（和不常见的）算法与技术。如果你是一位数据科学家，就会非常熟悉 NumPy、scikit-learn、pandas 以及其他库。这些库对数据科学工作至关重要。如果还没有真正理解数据科学，运用这些库也是开始数据科学工作的好方式。

在本书中，我们从零开始着手数据科学工作。这意味着为了获得更好的理解，我们需要自己亲手构建工具和实现算法。我花费了很多心思选择注释良好、简洁易读的实现范例。在大部分情形下，所建立的工具意义清晰但实用性有限，它们对规模较小的示例数据集运转良好，但对“网络级别”的数据集就束手无策了。

在全书中，我会向读者指出相应的库，用以将相应技术运用于大规模数据集，但本书中我们不会使用它们。

对学习数据科学，一直有这样一种积极的争论，即什么样的语言环境最好？许多人认为是统计语言 R。（我们说，他们错了。）还有一些人认为是 Java 或者 Scala。而我认为，Python 才是最佳选择！

对于学习和从事数据科学工作，Python 具有几大优势：

- 免费；
- 编程相对简单（尤其是也易于理解）；
- 具有很多数据科学相关的库。

我不敢说 Python 是我最爱的编程语言，因为的确存在其他一些更舒适、设计更棒、编程更有乐趣的语言。但是，每当着手一个新的数据科学项目时，我最终使用的是 Python；每当需要快速构建某个有效程序的原型时，我使用的是 Python；每当需要用简洁易懂的方式表达数据科学概念时，我使用的还是 Python。于是，本书也采用 Python。

但是，教授 Python 不是本书的目的（尽管通过学习本书你会学到一些 Python 知识）。本书会用一章快速介绍 Python 的重要特征，这些特征与本书目的紧密相关。倘若读者没有 Python 基础（或编程基础），那需要再补充阅读一些关于 Python 的入门指导。

本书数据科学导论的其余部分采取了类似的书写方式，在必要或需要阐明时才深入细节，否则省略细节留给读者自己去挖掘（或者在维基百科上查阅）。

过去我曾培训过许多数据科学家。不是每个人都会努力变成改变世界的明星级数据忍者，但所有人都通过培训成为了更棒的数据科学家。我越来越相信，任何拥有一定数学基础和编程技术的人，只要再匹配一些基本材料就可以从事数据科学工作。必需品是好奇心、勤奋工作的态度，还有本书。没错，就是本书！

本书排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。

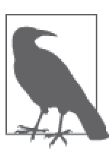
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 等宽粗体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

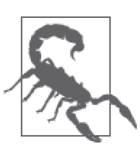
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注释。



该图标表示警告或警示。

示例代码的使用

本书的补充材料（示例代码、练习等）都可以从 GitHub 下载：https://github.com/joelgrus/

data-science-from-scratch。

本书提供代码的目的是帮你快速完成工作。一般情况下，你可以在你的程序或文档中使用本书中的代码，而不必取得我们的许可，除非你想复制书中很大一部分代码。例如，你在编写程序时，用到了本书中的几个代码片段，这不必取得我们的许可。但若将 O'Reilly 图书中的代码制成光盘并进行出售或传播，则需获得我们的许可。引用示例代码或书中内容来解答问题无需许可。将书中很大一部分的示例代码用于你个人的产品文档，这需要我们的许可。

如果你引用了本书的内容并标明版权归属声明，我们对此表示感谢，但这不是强制的。版权归属声明通常包括：标题、作者、出版社和 ISBN，例如："*Data Science from Scratch* by Joel Grus (O'Reilly). Copyright 2015 Joel Grus, 978-1-4919-0142-7"。

如果你认为你对示例代码的使用已经超出上述范围，或者你对是否需要获得示例代码的授权还不清楚，请随时联系我们：permissions@oreilly.com 。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体（https://www.safaribooksonline.com/enterprise/）、政府机构（https://www.safaribooksonline.com/government/）、教育机构（https://www.safaribooksonline.com/academic-public-library/）和个人，Safari Books Online 提供各种产品组合和灵活的定价策略（https://www.safaribooksonline.com/pricing/）。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社（https://www.safaribooksonline.com/our-library/）的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：
http://shop.oreilly.com/product/0636920033400.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

首先，我深深地感谢 Mike Loukides 接受我写作本书的提议（并对本书的篇幅提出了合理的建议）。其实他可以选择更轻松的方式，例如可以说："那个总发样章给我的是什么人？我该怎样拒绝他？"但他没有，我感激不尽！同样，我很感谢我的编辑 Marie Beaugureau，她在整个出版过程中给予我指导，并最终使本书呈现出比我独立完成更棒的状态。

如果我从未学习数据科学，怎么能写出这样一本书？如果没有 Dave Hsu、Igor Tatarinov、John Rauser 和 Farecast 群组其他人的影响，我不大可能学习数据科学（当时甚至还没有数据科学这个名称）。还要感谢免费大型公开课程项目 Coursera 的优秀教师们。

同样，我很感谢本书的试读者和评论者。Jay Fundling 找出了许多错误，并指出很多含混的表达，因而本书得到了很大的改善，非常感谢。Debashis Ghosh 全面检查了本书的统计学部分。本书原本表达了肯定 Python 否定 R 的看法，Andrew Musselman 建议淡化这种立场，我后来体会到这是金玉良言。我也非常感谢 Trey Causey、Ryan Matthew Balfanz、Loris Mularoni、Núria Pujol、Rob Jefferson、Mary Pat Campbell、Zach Geary 和 Wendy Grus

给我的宝贵反馈。当然，本书其余的问题，责任在我。

我非常感谢 Twitter 上的数据科学社区，它让我接触到很多新奇的概念，让我认识了很多大牛。我深感自己的欠缺，需要写一本书来弥补。（再次）特别感谢 Trey Causey，他（并非刻意地）提醒我在书中加一章线性代数的内容。同样感谢 Sean J. Taylor，他（并非刻意地）指出了数据处理一章中的若干重大缺漏。

最后，向 Ganga 和 Madeline 致以我无限的感恩。比写一本书更痛苦的事，莫过于和写这本书的人朝夕相对。本书得以完成，全赖家人的支持与鼓励。

第 1 章
导论

“数据！数据！！数据！！！”他不耐烦地咆哮着，“我不能做无米之炊！”

——阿瑟·柯南·道尔

1.1 数据的威力

生活中，数据无处不在。用户的每次点击，网站都会记录下来。你每时每刻的位置和速度，智能手机也会记录下来。“量化自我”生活方式的倡导者使用智能计步器记录心率、行动习惯、饮食习惯、睡眠方式。智能汽车记录驾驶习惯，智能家居设施记录生活习惯，智能购物设备记录购物习惯，等等。互联网是一个广袤的知识谱系，包括有无数交叉引用的百科全书，电影、音乐、赛讯、弹球机、模因、鸡尾酒等各种专业数据库，以及许多政府发布的多得让人理不清头绪的统计数据（某些还是比较真实的）。

在这些数据之中隐藏着无数问题的答案，这些问题从没有人提出过。让我们在这本书中一起学习如何找出这些问题。

1.2 什么是数据科学

有这样一个形容数据科学家的小笑话。数据科学家有什么特点呢？他们是计算机科学家中的统计专家，是统计专家中的计算机科学家（抱歉，笑话有点冷）。事实上，有些数据科学家的确是统计专家，而有些数据科学家则堪比软件工程师。有的数据科学家是机器学习专家，而也有一些数据科学家仅仅是这方面的菜鸟。有的数据科学家拥有博士学位，出版

过出色的学术作品，而有些数据科学家从不阅读论文（我都有点不好意思了）。总之，不管你如何定义数据科学家，总会找到某些反例来否定这样的定义。

然而，这并不会阻止我们尝试为数据科学家下定义。我们认为，数据科学家是能够从混乱数据中剥离出洞见的人。今天，世界各地有无数人着力于此。

举个例子，一个名叫 OkCupid 的约会网站为了给会员找到合适的对象，要求他们回答上千个问题。OkCupid 通过分析这些答案来确定你可以问哪些无伤大雅的问题，以此来猜测你和某个心仪之人初次约会时会有多大可能直接奔向三垒（http://blog.okcupid.com/index.php/the-best-questions-for-first-dates/）。

再举个例子，你在 Facebook 上需要填写家乡和居住地的信息。表面上看，网站是在帮助你的朋友更容易地找到你，联系你。但实际上，除此以外，网站还通过分析地理信息来研究全球移民模式（https://www.facebook.com/notes/facebook-data-science/coordinated-migration/10151930946453859），或者研究不同球队的粉丝分布情况（https://www.facebook.com/notes/facebook-data-science/nfl-fans-on-facebook/10151298370823859）。

另一个例子，一个名叫 Target 的大型零售商，会追踪消费者在线上线下的购买与互动数据（http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html），再从中分析，预测哪些客户怀孕了，以便向其推销合适的母婴商品。

最后，举一个奥巴马 2012 年竞选的例子。他的竞选团队雇用了很多数据科学家，专家们搜集选民的相关数据，通过数据挖掘识别不同的选民。他们通过实验的方法确定哪些选民需要更多的关注，并选择最有鼓舞性的拉票活动，把重心放在最能吸引选票的活动上。最后，奥巴马胜出，成功地第二次出任总统。人们普遍认为数据科学家功不可没。同时，这意味着数据分析在未来竞选中会扮演越来越重要的角色，一场数据竞争的硝烟弥漫开来，好戏刚刚开始。

现在，如果谈论竞争令你感到疲惫，那我们转而谈谈公益——数据科学家偶尔也会使用数据来帮助政府提高工作效率（http://www.marketplace.org/topics/tech/beyond-ad-clicks-using-big-data-social-good）、帮助流浪者（http://dssg.io/2014/08/20/paths-homelessness.html）、改善公共健康（https://plus.google.com/communities/109572103057302114737）等。当然，如果你能想出好方法来提高广告点击率，这对你的职业发展也不会有什么坏处。

1.3 激励假设：DataSciencester

恭喜！你被聘请来领导 DataSciencester 的数据科学工作。DataSciencester 是数据科学家们的社交网络。

虽然号称为数据科学家服务，DataSciencester 却从未践行数据科学任务（同样也从未构建

自己的产品）。当然，这是你的工作。在本书中，我们通过解决在工作中碰到的一个个问题，来学习数据科学的思想。我们有时会直接研究用户提供的数据，有时会研究用户与网站互动生成的数据，有时研究从我们自己设计的实验中获得的数据。

DataSciencester 拥有一种特别强烈的原创精神——“非我莫属”（Not-Invented-Here，NIH），即工作中使用到的工具必须自己亲手创建。这样完成工作之后，你会全面深入地理解数据科学。将来，你更可以将自己所学运用于更有前途的公司，或去解决任何有趣的问题。

欢迎加入，祝你好运！（周五可以穿牛仔裤上班，洗手间位于大厅右侧。）

1.3.1 寻找关键联系人

现在，你在 DataSciencester 开始第一天的工作。网络部的副总一直有一些关于客户的问题没有解决，以前找不到人帮忙，现在你来了，他特别高兴。

第一，他需要你识别出数据科学家中的“关键联系人”。因而他转给你 DataSciencester 所有用户的网络关系数据。（实际工作中，你需要的数据不会轻而易举地拿过来。我们在第 9 章专门讨论获取数据的方法。）

从整体上看，数据是一个包含所有用户的列表。列表的每个元素是一个字典（即一个 `dict`）。字典中包含了用户的 ID（即 `id`）和账号名（即 `name`。在这批数据中，`name` 与 `id` 的数字为谐音）：

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

同时，他也给了你用户的“友邻关系”数据列表。这个列表的元素是成对的 `id`：

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

比如说，元组 `(0, 1)` 表示 `id` 为 0 的数据科学家 Hero 和 `id` 为 1 的数据科学家 Dunn 是朋友。这种网络关系可以用图 1-1 来描述。

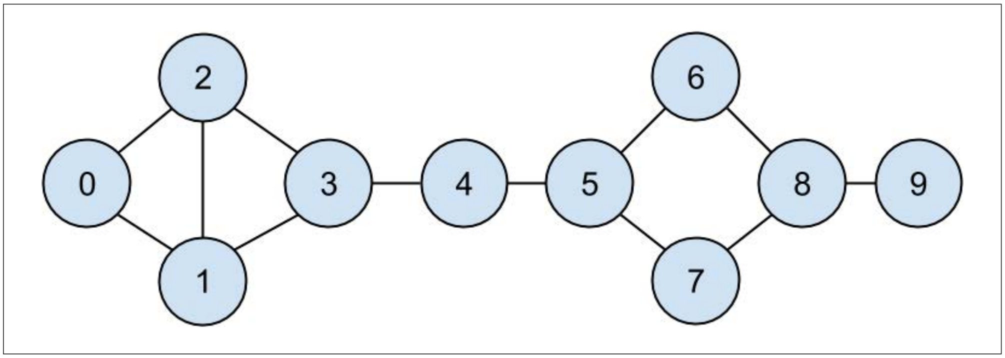


图 1-1：DataSciencester 网络

我们使用字典结构 dict 表示用户，因而可以方便地添加更多数据。

> 目前不要纠结代码细节，我们会在第 2 章指导你快速学习 Python。现在你只需要大致了解代码的功能。

比如，如果我们希望对每个用户增加一个朋友列表，首先需要对每个用户创建一个空列表：

```
for user in users:
    user["friends"] = []
```

再用 friendships 数据填充：

```
for i, j  in friendships:
    # 这能起作用是因为users[i]是id为i的用户
    users[i]["friends"].append(users[j]) # 把i加为j的朋友
    users[j]["friends"].append(users[i]) # 把j加为i的朋友
```

完成之后，每个用户的 dict 都会包含一个朋友列表，进而可以深入地研究朋友网络关系，比如提问“平均的联系数是多少”（即每位用户平均拥有几位朋友）。

首先计算出全部的联系数，这需要对所有用户的 friends 列表的长度求和：

```
def number_of_friends(user):
    """how many friends does _user_have?"""
    return len(user["friends"])                  # 列表friend_ids的长度

total_connections = sum(number_of_friends(user)
                        for user in users)       # 24
```

然后，将它除以用户个数：

```
from __future__ import division                    # 整数除法需要导入
num_users = len(users)                             # 列表users的长度
avg_connections = total_connections / num_users  # 2.4
```

这样就很容易看出，拥有最多联系的人就是拥有最多朋友数目的人。

因为用户不多，所以能很方便地按照朋友数的多少排序：

```
# 创建一个列表(user_id, number_of_friends)
num_friends_by_id = [(user["id"], number_of_friends(user))
                     for user in users]

sorted(num_friends_by_id,                                # 把它按照
       key=lambda (user_id, num_friends): num_friends, # num_friends
       reverse=True)                                     # 从最大到最小排序

# 每一对都是(user_id, num_friends)
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
#  (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

可将以上行为视为一种识别谁处在人际网络中心的方法。事实上，以上计算的是*度中心性*，是一种网络度量，如图 1-2 所示。

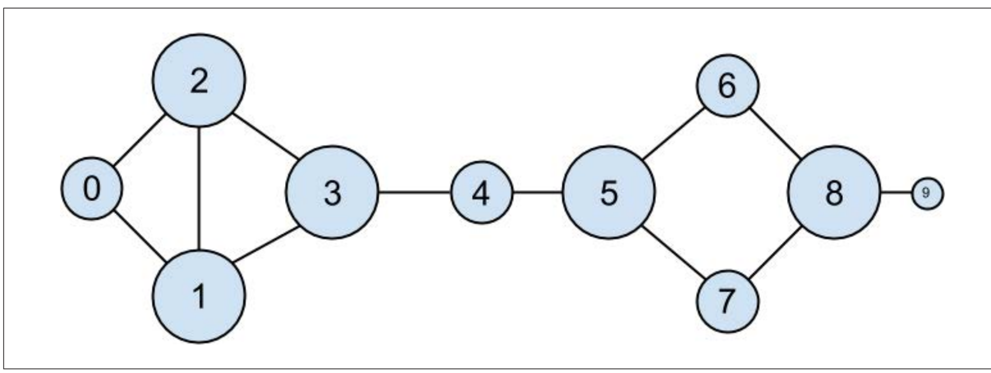


图 1-2：利用度计算 DataSciencester 的网络大小

度中心性简单易算，但不能总如你所愿。比如，在 DataSciencester 的网络中，Thor（id 为 4）只有两个联系数，Dunn（id 为 1）有三个。从网络关系图中看，直观上感觉 Thor 处于中心地位。第 21 章将考察网络关系的更多细节，探讨更多关于中心性的复杂概念，它们可能与直觉一致，也可能不一致。

1.3.2　你可能知道的数据科学家

当你正在填写新员工入职表格时，人力部门的副总来到你桌旁。她想鼓励数据科学家们多多联系，因而希望你设计一个“你可能知道的数据科学家”的提示函数。

你的直觉是用户可能会认识朋友的朋友。这不难计算：对某个用户，依次计算每个朋友的朋友，最后合并结果：

```
def friends_of_friend_ids_bad(user):
    # foaf是“朋友的朋友”的英文简写
    return [foaf["id"]
            for friend in user["friends"]   # 对每一位用户的朋友
            for foaf in friend["friends"]]  # 得到他们的朋友
```

当我们对 users[0]（Hero）调用上面这个函数时，它的结果显示为：

```
[0, 2, 3, 0, 1, 3]
```

因为 Hero 是他两位朋友的朋友，所以结果中包含两次用户 0。同时，因为用户 1 和用户 2 都是 Hero 的朋友，所以也包含在结果中。此外，由于用户 Chi 可以通过用户 1 和用户 2 与用户 0 联系，所以包含了他两次：

```
print [friend["id"] for friend in users[0]["friends"]]  # [1, 2]
print [friend["id"] for friend in users[1]["friends"]]  # [0, 2, 3]
print [friend["id"] for friend in users[2]["friends"]]  # [0, 1, 3]
```

有趣的是，人们可以通过朋友的朋友相互认识。受此启发，我们也许可以设计一个共同的朋友，由他来表示朋友的计数。同时，为了排除那些已经成为朋友的用户，我们需要设计一个辅助函数来实现这个功能：

```
from collections import Counter                      # 默认未加载

def not_the_same(user, other_user):
    """two users are not the same if they have different ids"""
    return user["id"] != other_user["id"]

def not_friends(user, other_user):
    """other_user is not a friend if he's not in user["friends"];
    that is, if he's not_the_same as all the people in user["friends"]"""
    return all(not_the_same(friend, other_user)
               for friend in user["friends"])

def friends_of_friend_ids(user):
    return Counter(foaf["id"]
                   for friend in user["friends"]   # 对我的每一位朋友
                   for foaf in friend["friends"]   # 计数他们的朋友
                   if not_the_same(user, foaf)     # 既不是我
                   and not_friends(user, foaf))    # 也不是我的朋友

print friends_of_friend_ids(users[3])             # Counter({0: 2, 5: 1})
```

这个输出结果正确无误地说明了 Chi（id 为 3）和 Hero（id 为 0）有两个共同的朋友，和 Clive（id 为 5）有一个共同的朋友。

出于一个数据科学家的直觉，你可能会喜欢结交有共同兴趣的人（这个例子很好地展示了

数据科学家的专业技能）。咨询之后，你设计出如下列表，每个元素都是成对数据（user_id, interest）：

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

例如，Thor（id 为 4）与 Devin（id 为 7）没有共同的朋友，但他们对机器学习都感兴趣。

如果需要找出对某种事物有共同爱好的用户，很容易设计出相应的函数：

```
def data_scientists_who_like(target_interest):
    return [user_id
            for user_id, user_interest in interests
            if user_interest == target_interest]
```

但是，上面的算法每次搜索都需要遍历整个兴趣列表，如果用户很多或者用户的兴趣很多（或我们只是想多进行一些查找），这种算法的时间和空间成本会很大，因此最好能建立一个从兴趣到用户的索引直接搜索：

```
from collections import defaultdict

# 键是interest,值是带有这个interest的user_id的列表
user_ids_by_interest = defaultdict(list)

for user_id, interest in interests:
    user_ids_by_interest[interest].append(user_id)
```

以及另一个从用户到兴趣的索引：

```
# 键是user_id,值是对那些user_id的interest的列表
interests_by_user_id = defaultdict(list)
for user_id, interest in interests:
    interests_by_user_id[user_id].append(interest)
```

现在，给定一个用户，可以方便地找到与他共同爱好最多的用户：

- 迭代这个用户的兴趣；

- 针对这个用户的每一种兴趣，寻找这种兴趣的其他用户，并迭代；
- 记录每一个用户在循环中出现的次数。

```
def most_common_interests_with(user):
    return Counter(interested_user_id
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"])
```

接下来，结合共同的朋友和共同的兴趣，可以建立一个更丰富的功能“你应该知道的数据科学家”，这类应用我们将在第 22 章探讨。

1.3.3 工资与工作年限

当你正准备去吃午饭时，公共关系部门的副总突然来到你的办公室，问你能不能给他提供一些关于数据科学家的收入的有趣数据。当然，大家对工资数据都很敏感，他想办法搞出了一份匿名文件，其中包含每位用户的工资（salary）和作为数据科学家的工作年限（tenure）：

```
salaries_and_tenures = [(83000, 8.7), (88000, 8.1),
                        (48000, 0.7), (76000, 6),
                        (69000, 6.5), (76000, 7.5),
                        (60000, 2.5), (83000, 10),
                        (48000, 1.9), (63000, 4.2)]
```

很自然地，第一步是绘制数据（第 3 章有详细介绍），如图 1-3 所示。

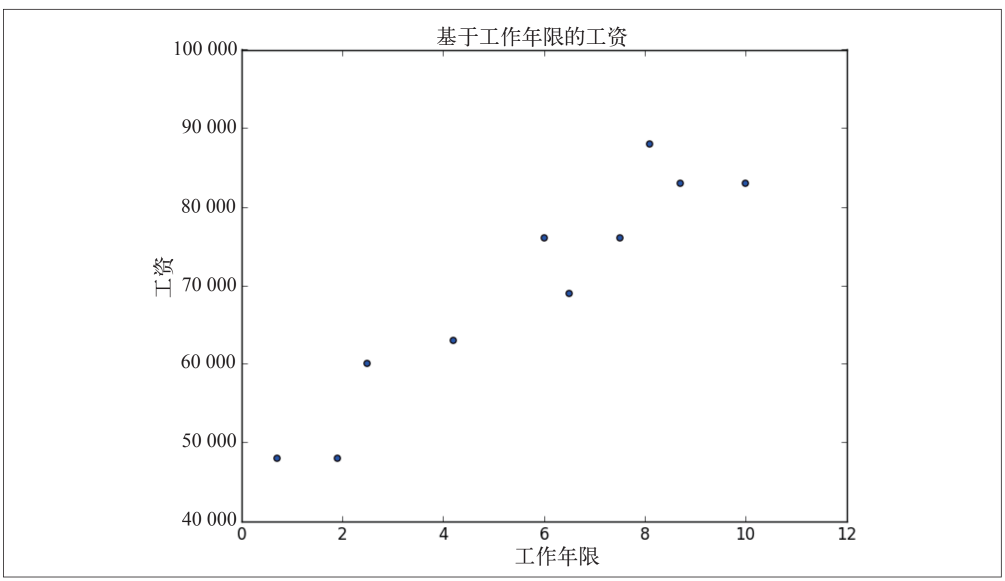


图 1-3：基于工作年限的工资图

图中所示的关系很明显，工作年限越长的人收入越高。接下来需要考虑的是如何将这个结果转化成更有趣的事实。你首先想到了考察一下大家的年均收入：

```
# 键是year,值是对每一个tenure的salary的列表
salary_by_tenure = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    salary_by_tenure[tenure].append(salary)

# 键是year,每个值是相应tenure的平均salary
average_salary_by_tenure = {
    tenure : sum(salaries) / len(salaries)
    for tenure, salaries in salary_by_tenure.items()
}
```

实际上，任何两个用户都没有相同的工作年限，所以上述计算结果作用有限，它仅仅说明了每个用户独立的收入：

```
{0.7: 48000.0,
 1.9: 48000.0,
 2.5: 60000.0,
 4.2: 63000.0,
 6: 76000.0,
 6.5: 69000.0,
 7.5: 76000.0,
 8.1: 88000.0,
 8.7: 83000.0,
 10: 83000.0}
```

一个更有意义的计算方式是把用户的工作年限分组：

```
def tenure_bucket(tenure):
    if tenure < 2:
        return "less than two"
    elif tenure < 5:
        return "between two and five"
    else:
        return "more than five"
```

再将每组的工资合并：

```
# 键是tenure bucket,值是相应bucket的salary的列表
salary_by_tenure_bucket = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    bucket = tenure_bucket(tenure)
    salary_by_tenure_bucket[bucket].append(salary)
```

最后，计算每个分组的平均工资：

```
# 键是tenure bucket,值是对那个bucket的average salary
average_salary_by_bucket = {
```

```
    tenure_bucket : sum(salaries) / len(salaries)
    for tenure_bucket, salaries in salary_by_tenure_bucket.iteritems()
}
```

这是更有趣的结果：

```
{'between two and five': 61500.0,
 'less than two': 48000.0,
 'more than five': 79166.66666666667}
```

现在，你得到结论："有 5 年以上工作年限的数据科学家比同行新人的收入高 65%。"

但是，我们的选择是任意的。我们原本希望说明的是，平均看来，更多的工作年限意味着更多的工资收入。为了得到更多有趣的结论，我们可以预测一些未知年限的工资。我们将在第 14 章探讨这个想法。

1.3.4 付费账户

当你回到桌旁时，收益部门的副总在等你。他想更好地了解哪些用户会为账户付费，哪些用户不会。（他知道他们的名字，但这些信息不是特别有用。）

你注意到在工作年限和付费账户之间似乎存在一种对应关系：

```
0.7 paid
1.9 unpaid
2.5 paid
4.2 unpaid
6   unpaid
6.5 unpaid
7.5 unpaid
8.1 unpaid
8.7 paid
10  paid
```

那些新手和资历很深的用户倾向于付费，而那些具有中等工作年限的用户则倾向于不付费。

由此，如果你打算创建一个模型——尽管这点数据对创建模型肯定是不够的——你会试图对新手和资深用户预测"付费"，而对具有中等工作年限的用户预测"不付费"：

```
def predict_paid_or_unpaid(years_experience):
  if years_experience < 3.0:
    return "paid"
  elif years_experience < 8.5:
    return "unpaid"
  else:
    return "paid"
```

当然，我们会完全紧盯这个切入口。

利用更多的数据（和更多的数学计算），我们可以基于用户的工作年限来预测用户付费的可能性。我们会在第 16 章研究这类问题。

1.3.5　兴趣主题

当你正准备结束第一天的工作时，内容策略部门的副总来向你要数据，想了解什么样的主题更令用户感兴趣，以便据此规划他的博客日历。你已经有了来自友邻推荐项目的原始数据：

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

一种简单（但并不激动人心）的方法是仅仅数一下兴趣词汇的个数：

1. 小写每一种兴趣（因为不同的用户不一定会大写他们的兴趣）；
2. 把它划分为单词；
3. 数一数结果。

用下面的代码：

```
words_and_counts = Counter(word
                           for user, interest in interests
                           for word in interest.lower().split())
```

列出出现一次以上的词汇是很容易的：

```
for word, count in words_and_counts.most_common():
    if count > 1:
        print word, count
```

这给出了你所期待的结果（除非你想让“scikit-learn”分化成两个词，那样就不会得到预期的结果了）：

```
learning 3
java 3
```

```
python 3
big 3
data 3
hbase 2
regression 2
cassandra 2
statistics 2
probability 2
hadoop 2
networks 2
machine 2
neural 2
scikit-learn 2
r 2
```

我们会在第 20 章学习更多从数据中提取主题的复杂方法。

1.4　展望

第一天的工作非常成功！你肯定很累，赶快趁没人继续问问题时回家吧。明天是新员工培训，所以今晚好好休息一下。（当然啦，你还没接受培训就已经工作一整天了！明天去找 HR 吧。）

第2章

Python速成

> 难以置信，25 年以来 Python 始终广受追捧。
>
> ——迈克尔 · 佩林

DataSciencester 要求所有新员工接受入职培训，其中最有趣的部分当属 Python 速成。

本章不是完整的 Python 培训教程，而仅强调其中对我们最重要的部分（其中有些部分并非 Python 培训教程的重点）。

2.1 基础内容

2.1.1 Python获取

可以从 python.org（https://www.python.org/）网站下载 Python。但是对于没有安装过 Python 的读者，特别推荐安装 Anaconda 版本（https://www.continuum.io/downloads），这个版本涵盖了数据科学工作需要用到的大多数库。

当我写本书时，Python 的最新版本是 3.4。但是，在 DataSciencester，我们使用一个旧的可靠版本，Python 2.7。Python 3 无法向后兼容 Python 2，因此很多重要功能只能在 Python 2.7 上良好运行。数据科学社区中，2.7 版本一直是主流，我们也和它保持一致。请设法安装这个版本。

如果你没有 Anaconda 版本，那么需要安装 pip（https://pypi.python.org/pypi/pip），这是一个 Python 包管理器，可以用来方便地安装第三方包（其中有些是我们必需的）。IPython

（http://ipython.org/）也不错，操作界面更友好。

（如果你已经安装了 Anaconda 版本，它会很好地和 pip 与 IPython 兼容。）

运行：

```
pip install ipython
```

如果碰到难解的错误信息，可以在网上搜索解决办法。

2.1.2 Python之禅

Python 的设计原则（http://legacy.python.org/dev/peps/pep-0020/）有着禅宗的意味，你输入 **import this** 就能在 Python 解释器中一窥玄机。

讨论最多的原则之一是：

> 应该有一个——且最好只有一个——明显的方式去完成工作。

按照这种“明显”的方式（对一个新人来说可能根本不明显）编写的代码常常被称为具有“Python 风格”。尽管本书不是专门介绍 Python 的书，但我们时不时地会比较 Python 风格和非 Python 风格的方式在解决相同问题时的差异，而且你常常会发现 Python 风格是更好的解决方式。

2.1.3 空白形式

许多编程语言用大括号分隔代码块。Python 使用缩进的方式：

```
for i in [1, 2, 3, 4, 5]:
    print i                    # "for i"程序块的第一行
    for j in [1, 2, 3, 4, 5]:
        print j                # "for j"程序块的第一行
        print i + j            # "for j"程序块的最后一行
    print i                    # "for i"程序块的最后一行
print "done looping"
```

这样会使 Python 代码非常易读，但这也意味着你需要非常小心格式。系统会省略方括号和圆括号中的空白，这对冗长的计算非常有用：

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

并能使代码更易读：

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

easier_to_read_list_of_lists = [ [1, 2, 3],
```

```
                        [4, 5, 6],
                        [7, 8, 9] ]
```

你可以用反斜线代表一个语句在下一行续写，尽管这个例子并不常见：

```
two_plus_three = 2 + \
                 3
```

空白形式的一个后果是很难将代码复制并粘贴到 Python 的 shell。比如，如果你粘贴代码：

```
for i in [1, 2, 3, 4, 5]:

    # 注意这个空行
    print i
```

在粘贴入一般的 Python shell 后，会得到以下提示：

```
IndentationError: expected an indented block
```

因为解释器认为空行表示 for 循环的终结。

IPython 有一个奇妙的函数 %paste，可以正确地复制剪贴板上的内容，包括空白在内。这个函数足以成为选择使用 IPython 的好理由。

2.1.4 模块

Python 的某些特征默认不加载，包含了语言本身的部分特征，也包含了需读者自行下载的第三方特征。为了使用这些特征，你需要导入包含这些特征的模块。

一种方式是简单地导入模块本身：

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

这里的 re 代表包含了处理正则表达式需要的函数与常量的模块。输入 import 之后，你可以通过加前缀 re. 来直接调用模块中的函数。

如果你的代码中已经有了不同的 re，可以使用别名：

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

如果模块名冗长，或者你需要敲很多字符，那不妨试试这个方法。比如，你想对数据用模块 matplotlib 来进行可视化，标准转换如下：

```
import matplotlib.pyplot as plt
```

如果你需要一个模块中的一些特定值，可以显式导入，直接使用，不必提前获取权限，如

下所示：

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

如果你是破坏者，可以将模块的全部内容导入命名空间，这也许会不加提示地覆盖原先定义的变量：

```
match = 10
from re import *      # 呃,re有一个match函数
print match           # "<function re.match>"
```

但是，不是破坏者就别这样做。

2.1.5 算法

Python 2.7 默认整除，因此 `5 / 2` 等于 `2`。但是这并不总是我们想要的，因此文件常常需要这样开始：

```
from __future__ import division
```

这样 `5 / 2` 等于 `2.5`。本书中的所有示例程序采用新除法。在少数需要整除的情形下，我们可以通过双斜线 `5 // 2` 表示。

2.1.6 函数

函数是一种规则，输入零或者其他数，得到相应的输出。在 Python 中，我们用 `def` 定义函数：

```
def double(x):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2
```

Python 函数是*第一类函数*，第一类函数意味着可以将它们赋给其他变量，也可以像其他参数一样传递给函数：

```
def apply_to_one(f):
    """calls the function f with 1 as its argument"""
    return f(1)

my_double = double             # 指向之前定义的函数
x = apply_to_one(my_double)    # 等于2
```

也很容易生成简短的匿名函数，或者 lambda：

```
y = apply_to_one(lambda x: x + 4)     # 等于5
```

你可以将 lambda 赋给变量，尽管大部分人会建议你用 `def`：

```
another_double = lambda x: 2 * x      # 别这么做
def another_double(x): return 2 * x   # 要这么做
```

默认参数也可以赋值给函数参数，当你需要默认值以外的值时需要具体说明：

```
def my_print(message="my default message"):
    print message
my_print("hello")      # 打印"hello"
my_print()             # 打印"my default message"
```

有时候通过名字指定参数会有用：

```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # 返回5
subtract(0, 5)  # 返回-5
subtract(b=5)   # 和前一句一样
```

我们可以生成很多很多函数。

2.1.7　字符串

字符串可以用单引号或者双引号标注分隔出来（但引号需要配对，即单引号对单引号，双引号对双引号）：

```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

Python 用反斜线来为特殊字符编码。比如：

```
tab_string = "\t"    # 表示tab字符
len(tab_string)      # 是1
```

如果你希望反斜线仅仅代表反斜线本身（你在 Windows 系统中的文件夹或者正则表达式中也许会遇到），那么可以使用命令 `r""` 生成一个原始的字符串：

```
not_tab_string = r"\t"    # 表示字符'\'和't'
len(not_tab_string)       # 是2
```

你可以通过三重［两重］引号来生成多行的字符串：

```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""
```

2.1.8 异常

当发生意外时，Python 会报出异常。如果不处理，异常会引起程序崩溃。你可以用 `try` 和 `except` 来解决：

```
try:
    print 0 / 0
except ZeroDivisionError:
    print "cannot divide by zero"
```

在许多语言中，异常意味着坏情形。但在 Python 中，你不必害怕遇见异常，只要能写出清晰的代码，我们时而会这样做。

2.1.9 列表

Python 中最基本的数据结构是列表（`list`）。一个列表是一个有序的集合。（这和其他语言中的数组概念类似，但增加了函数功能。）

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list)      # 等于3
list_sum    = sum(integer_list)      # 等于6
```

你可以通过方括号对列表的第 n 个元素读值和赋值：

```
x = range(10)   # 是列表[0, 1,    ..., 9]
zero = x[0]     # 等于0,列表是0-索引的
one = x[1]      # 等于1
nine = x[-1]    # 等于9,最后一个元素的Python惯用法
eight = x[-2]   # 等于8,倒数第二个元素的Pyhton惯用法
x[0] = -1       # 现在x是[-1, 1, 2, 3,    ..., 9]
```

你也可以用方括号来“切取”列表：

```
first_three   = x[:3]               # [-1, 1, 2]
three_to_end = x[3:]                # [3, 4, ..., 9]
one_to_four = x[1:5]                # [1, 2, 3, 4]
last_three = x[-3:]                 # [7, 8, 9]
without_first_and_last = x[1:-1]    # [1, 2, ..., 8]
copy_of_x = x[:]                    # [-1, 1, 2, ..., 9]
```

Python 可以通过操作符 `in` 确认列表成员：

```
1 in [1, 2, 3]    # True
0 in [1, 2, 3]    # False
```

确认每次都会遍历列表元素。这意味着除非列表很小，否则就不应该进行确认（除非你不在乎确认需要花费多长时间）。

将列表串连起来很容易：

```
x = [1, 2, 3]
x.extend([4, 5, 6])   # x现在是[1,2,3,4,5,6]
```

如果不希望改变原序列 x，你可以使用列表加法：

```
x = [1, 2, 3]
y = x + [4, 5, 6]    # y是[1, 2, 3, 4, 5, 6];x是不变的
```

更常见的做法是，一次在原列表上只增加一项：

```
x = [1, 2, 3]
x.append(0)       # x现在是[1, 2, 3, 0]
y = x[-1]         # 等于0
z = len(x)        # 等于4
```

如果你知道列表中元素的个数，可以方便地从中提取值：

```
x, y = [1, 2]    # 现在x是1,y是2
```

不过，如果等式两端元素个数不同，会报出提示 ValueError。

如果你希望忽略某些值，常见的选择是使用下划线：

```
_, y = [1, 2]    # 现在y==2,不用关心第一个元素是什么
```

2.1.10 元组

元组是列表的亲表哥。你对列表做的很多操作都可以对元组做，但不包括修改。元组通过圆括号（或者什么都不加）而不是方括号来给出具体的描述：

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3     # my_list现在是[1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print "cannot modify a tuple"
```

元组是通过函数返回多重值的便捷方法：

```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)     # 等于(5,6)
s, p = sum_and_product(5, 10)  # s是15,p是50
```

元组（和列表）都可以进行多重赋值（multiple assignment）：

```
x, y = 1, 2     # 现在x是1,y是2
x, y = y, x     # Python风格的互换变量,现在x是2,y是1
```

2.1.11 字典

Python 中的另一种基本数据结构是字典，它将值与键联系起来，让我们可以通过键快速找到对应值：

```
empty_dict = {}                         # Python风格
empty_dict2 = dict()                    # 更少的Python风格
grades = { "Joel" : 80, "Tim" : 95 }    # 字典
```

你也可以通过方括号查找键的值：

```
joels_grade = grades["Joel"]            # 等于80
```

如果你找的键不在字典中，会得到 KeyError 报错：

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print "no grade for Kate!"
```

你可以用 in 确认键的存在：

```
joel_has_grade = "Joel" in grades     # 正确
kate_has_grade = "Kate" in grades     # 错误
```

如果查找的键在字典中不存在，字典可以通过方法 get 返回默认值（而非报出异常）：

```
joels_grade = grades.get("Joel", 0)   # 等于80
kates_grade = grades.get("Kate", 0)   # 等于0
no_ones_grade = grades.get("No One")  # 默认的默认值为None
```

你可以通过方括号来为键值对赋值：

```
grades["Tim"] = 99                    # 替换了旧的值
grades["Kate"] = 100                  # 增加了第三个记录
num_students = len(grades)            # 等于3
```

我们常常使用字典作为代表结构数据的简单方式：

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"]
}
```

除了查找特定的键，我们还可以查找所有值：

```
tweet_keys   = tweet.keys()      # 键的列表
tweet_values = tweet.values()    # 值的列表
tweet_items  = tweet.items()     # (键, 值)元组的列表

"user" in tweet_keys             # True,使用慢速的列表
"user" in tweet                  # 更符合Python惯用法,使用快速的字典
"joelgrus" in tweet_values       # True
```

字典的键不可改变，尤其是不能将列表作为键。如果你需要一个多维的键，应该使用元组或设法把键转换成字符串。

1. defaultdict

假设你需要计算某份文件中的单词数目。一个明显的方式是，建立一个键是单词、值是单词出现次数的字典。每次你查到一个单词，如果字典中存在这个词，就在该词的计数上增加 1，如果字典中没有这个词，就把这个词增加到这个字典中：

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

当查找缺失值碰到异常报出时，你可以遵循“与其瞻前顾后，不如果断行动”（Forgiveness is better than permission）的原则，果断处理异常：

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

第三种方法是使用 get，这种处理缺失值的方法比较优雅：

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

以上三种方法都略显笨拙，这是 defaultdict 的意义之所在。一个 defaultdict 相当于一个标准的字典，除了当你查找一个没有包含在内的键时，它用一个你提供的零参数函数建立一个新的键，并为它的值增加 1。为了使用 defaultdict，你需要将其从集合中导出：

```
from collections import defaultdict

word_counts = defaultdict(int)          # int()生成0
```

```
for word in document:
    word_counts[word] += 1
```

这对列表、字典或者你自己的函数都有用：

```
dd_list = defaultdict(list)             # list()生成一个空列表
dd_list[2].append(1)                    # 现在dd_list包含{2:[1]}

dd_dict = defaultdict(dict)             # dict()产生一个新字典
dd_dict["Joel"]["City"] = "Seattle"     # { "Joel" : { "City" : Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                       # 现在dd_pair包含{2: [0,1]}
```

当我们用字典“收集”某些键对应的结果，并且不希望每次查找某键是否存在都遍历一遍的时候，defaultdict 非常有用。

2. Counter

一个计数器将一个序列的值转化成一个类似于整型的标准字典（即 defaultdict(int)）的键到计数的对象映射。我们主要用它来生成直方图：

```
from collections import Counter
c = Counter([0, 1, 2, 0])        # c是(基本的) { 0 : 2, 1 : 1, 2 : 1 }
```

这给我们提供了一个用来解决单词计数问题的很简便的方法：

```
word_counts = Counter(document)
```

一个 Counter 实例带有的 most_common 方法的例子如下：

```
# 打印10个最常见的词和它们的计数
for word, count in word_counts.most_common(10):
    print word, count
```

2.1.12　集合

另一种数据结构是集合（set)，它表示为一组不同的元素：

```
s = set()
s.add(1)        # s现在是1
s.add(2)        # s现在是{1,2}
s.add(2)        # s还是{1,2}
x = len(s)      # 等于2
y = 2 in s      # 等于True
z = 3 in s      # 等于False
```

我们使用集合的原因主要有两个。第一个是集合上有一种非常快速的操作：in。如果我们有大量的项目，希望对它的成分进行测试，那么使用集合比使用列表要合适得多：

```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"]
```

```
"zip" in stopwords_list     # False,但需要检查每个元素

stopwords_set = set(stopwords_list)
"zip" in stopwords_set      # 非常快地检查
```

第二个原因是便于在一个汇总中寻找其中离散的项目：

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)            # 6
item_set = set(item_list)             # {1, 2, 3}
num_distinct_items = len(item_set)    # 3
distinct_item_list = list(item_set)   # [1, 2, 3]
```

我们使用 `set` 的频率要远低于 `dict` 和 `list`。

2.1.13　控制流

和大多数编程语言一样，你可以用 `if` 语句来执行一种有条件的行动：

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

也可以在一行语句中使用 if-then-else，我们有时候需要这么做：

```
parity = "even" if x % 2 == 0 else "odd"
```

Python 也有 `while` 循环：

```
x = 0
while x < 10:
    print x, "is less than 10"
    x += 1
```

尽管我们更常用 `for` 和 `in`：

```
for x in range(10):
    print x, "is less than 10"
```

如果你需要更复杂的逻辑表达式，可以使用 `continue` 和 `break`：

```
for x in range(10):
    if x == 3:
        continue       # 直接进入下次迭代
    if x == 5:
        break          # 完全退出循环
    print x
```

上面的语句会打印 0、1、2 和 4。

2.1.14 真和假

Python 的布尔数除了首字母大写之外，其他用法和大多数别的语言类似：

```
one_is_less_than_two = 1 < 2        # 等于True
true_equals_false = True == False   # 等于False
```

Python 使用 None 来表示一个不存在的值，它类似别的语言中的 null：

```
x = None
print x == None    # 打印True,但这并非Python的惯用法
print x is None    # 打印True,符合Python的惯用法
```

Python 可以使用任何可被认为是布尔数的值。下面这些都是“假”（Falsy）：

- False
- None
- [ ]（一个空 list）
- { }（一个空 dict）
- ""
- set()
- 0
- 0.0

还有很多值可作为真（True）来处理。这样你可以很容易地使用 if 语句来对空列表、空字符串或空字典等进行测试。有时候，如果你没有意识到这种行为，会引入一些微妙的 bug：

```
s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""
```

另一种简单的方法是：

```
first_char = s and s[0]
```

这是因为第一个值为“真”时，and 运算符返回它的第二个值，否则返回第一个值。类似地，如果 x 的取值可能是一个数也可能是 None，那么以下代码的结果就必然会是一个数字：

```
safe_x = x or 0
```

Python 还有一个 all 函数，它的取值是一个列表，当列表的每个元素都为真时返回 True。

Python 还有一个 any 函数，当取值的列表中至少有一个元素为真时，它返回 True：

```
all([True, 1, { 3 }])    # True
all([True, 1, {}])       # False,{}为假
any([True, 1, {}])       # True
all([])                  # True,列表里没有假的元素
any([])                  # False,列表里没有真的元素
```

2.2　进阶内容

现在我们来看一些比较高级的 Python 特性，这些特性对开展数据工作特别有用。

2.2.1　排序

每个 Python 列表都有一个 sort 方法可以恰当地排序。如果你不想弄乱你的列表，可以使用 sorted 函数，它会返回一个新列表：

```
x = [4,1,2,3]
y = sorted(x)       # 结果是[1,2,3,4],但x没有变
x.sort()            # x变为[1,2,3,4]
```

默认情况下，sort（和 sorted）基于元素之间的朴素比较从最小值到最大值对列表进行排序。

如果你想把元素按从最大值到最小值进行排序，可以指定参数 reverse=True。除了比较元素本身，你还可以通过指定键来对函数的结果进行比较：

```
# 通过绝对值对列表元素从最大到最小排序
x = sorted([-4,1,-2,3], key=abs, reverse=True) # 是[-4,3,-2,1]

# 从最高数到最低数排序单词和计数
wc = sorted(word_counts.items(),
            key=lambda (word, count): count,
            reverse=True)
```

2.2.2　列表解析

我们有时可能会想把一个列表转换为另一个列表，例如只保留其中一些元素，或更改其中一些元素，或者同时做这两种变动。可以执行这种操作的 Python 技巧叫作*列表解析*（list comprehension）：

```
even_numbers = [x for x in range(5) if x % 2 == 0]     # [0, 2, 4]
squares      = [x * x for x in range(5)]               # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]           # [0, 4, 16]
```

类似地，你也可以把列表转换为字典或集合：

```
square_dict = { x : x * x for x in range(5) }    # { 0:0, 1:1, 2:4, 3:9, 4:16}
square_set  = { x * x for x in [1, -1] }         # { 1 }
```

如果你不需要来自原列表中的值，常规的方式是使用下划线作为变量：

```
zeroes = [0 for _ in even_numbers]    # 和even_numbers有相同的长度
```

列表解析可以包括多个 for 语句：

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]    # 100个对(0,0) (0,1) ... (9,8), (9,9)
```

其中后面的 for 语句可以使用前面的 for 语句的结果：

```
increasing_pairs = [(x, y)                       # 只考虑x < y的对
                    for x in range(10)           # range(lo, hi) 与之相等
                    for y in range(x + 1, 10)]   # [lo, lo + 1, ..., hi - 1]
```

我们会经常用到列表解析。

2.2.3 生成器和迭代器

列表的一个问题是它很容易变得非常大。range(1000000) 能创建一个有 100 万个元素的列表：如果你需要每次只处理其中一个元素，这将会是极大的资源浪费（或会导致内存不足）；如果你只需要前面的几个值，那对整个列表都进行计算也是一种浪费。

生成器（generator）是一种可以对其进行迭代（对我们来说，通常使用 for 语句）的程序，但是它的值只按需延迟（lazily）产生。

创建生成器的一种方法是使用函数和 yield 运算符：

```
def lazy_range(n):
    """a lazy version of range"""
    i = 0
    while i < n:
        yield i
        i += 1
```

下面的循环会每次消耗一个 yield 值直到一个也不剩：

```
for i in lazy_range(10):
    do_something_with(i)
```

（Python 确实有一个和 lazy_range 一样的函数，叫作 xrange，并且在 Python 3 中，range 函数本身就是延迟的。）这意味着，你甚至可以创建一个无限的序列：

```
def natural_numbers():
    """returns 1, 2, 3, ..."""
```

```
    n = 1
    while True:
        yield n
        n += 1
```

尽管在没有使用某种 break 逻辑语句时，你不应该做这种迭代的。

> 延迟的缺点是，你只能通过生成器迭代一次。如果需要多次迭代某个对象，你就需要每次都重新创建一个生成器，或者使用列表。

第二种创建生成器的方法是使用包含在圆括号中的 for 语句解析：

```
lazy_evens_below_20 = (i for i in lazy_range(20) if i % 2 == 0)
```

前面提过，每个 dict 都有一个 items() 方法可以返回它的键值对的列表。更常见的做法是使用 iteritems() 方法：当我们在列表上迭代的时候它延迟 yield 为每次一个键值对。

2.2.4 随机性

当我们学习数据科学时，会经常需要生成随机数。可以使用 random 模块生成随机数：

```
import random

four_uniform_randoms = [random.random() for _ in range(4)]

# [0.8444218515250481,
#  0.7579544029403025,
#  0.420571580830845,
#  0.25891675029296335]
# random.random()生成在0-1之间均匀分布的随机数,是最常用的随机函数
```

random 模块实际上生成的是基于一种内部状态的确定性的伪随机数。如果你想得到可复生的结果，可以用 random.seed 生成随机数种子：

```
random.seed(10)          # 设置随机数种子为10
print random.random()    # 0.57140259469
random.seed(10)          # 重设随机数种子为10
print random.random()    # 再次得到0.57140259469
```

有时候我们用 random.randrange 生成随机数，它会取 1 到 2 个参数，并从对应的 range() 函数随机选择一个元素返回：

```
random.randrange(10)    # 从range(10) = [0, 1, ..., 9]中随机选取
random.randrange(3, 6)  # 从range(3, 6) = [3, 4, 5]中随机选取
```

还有其他一些比较方便的方法，如 Random.shuffle 可随机地重排列表中的元素：

```
up_to_ten = range(10)
random.shuffle(up_to_ten)
print up_to_ten
# [2, 5, 1, 9, 7, 3, 8, 6, 4, 0],(你的结果可能不同)
```

如果你需要从列表中随机取一个元素，可以使用 random.choice：

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"])    # 对我来说是"Bob"
```

如果你需要不替换地（即不重复地）随机选择一个元素的样本，可以使用 random.sample：

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6)  # [16, 36, 10, 6, 25, 9]
```

选择一个允许替换的（即允许重复的）元素样本，只需多次调用 random.choice 即可：

```
four_with_replacement = [random.choice(range(10))
                         for _ in range(4)]
# [9, 4, 4, 2]
```

2.2.5　正则表达式

正则表达式提供了一种搜索文本的方法。它超乎想象地有用，但同时也相当复杂，以至于需要专门的书籍来讲解。之后的内容会频繁涉及正则表达式，届时我们再详述，这里只给出 Python 中如何使用正则表达式的例子：

```
import re

print all([                                 # 所有这些语句都为true,因为
    not re.match("a", "cat"),               # * 'cat'不以'a'开头
    re.search("a", "cat"),                  # * 'cat'里有一个字符'a'
    not re.search("c", "dog"),              # * 'dog'里没有字符'c'
    3 == len(re.split("[ab]", "carbs")),    # * 分割掉a,b,剩余长度为3
    "R-D-" == re.sub("[0-9]", "-", "R2D2")  # 用虚线进行位的替换
    ] # 打印True
```

2.2.6　面向对象的编程

就像许多语言一样，Python 允许你定义类（class）。类可以封装对象和函数来对它们进行操作。有时候我们会用类来使代码更加干净整洁。解释类的用法的最简单方式可能是构建一个有超多注释的例子。

假设没有内置的 Python 集合，那我们可能会想到去创建自己的 Set 类。

我们创建的类会有什么样的行为呢？给定一个 Set，我们需要能在其中加入（add）项目，移除（remove）项目，以及检查其中是否包含（contains）某个值。我们把这些功能创建

为成员（member）函数，意思是我们可以通过在 Set 对象后面加点（.）来访问它们：

```
# 按惯例,我们给下面的类起个PascalCase的名字
class Set:

    # 这些是成员函数
    # 每个函数都取第一个参数"self"(另一种惯例)
    # 它表示所用到的特别的集合对象

    def __init__(self, values=None):
        """This is the constructor.
        It gets called when you create a new Set.
        You would use it like
        s1 = Set()          # 空集合
        s2 = Set([1,2,2,3])  # 用值初始化"""

        self.dict = {} # Set的每一个实例都有自己的dict属性
                       # 我们会用这个属性来追踪成员关系
        if values is not None:
            for value in values:
                self.add(value)

    def __repr__(self):
        """this is the string representation of a Set object
        if you type it at the Python prompt or pass it to str()"""
        return "Set: " + str(self.dict.keys())

    # 通过成为self.dict中对应值为True的键,来表示成员关系
    def add(self, value):
        self.dict[value] = True

    # 如果它在字典中是一个键,那么在集合中就是一个值
    def contains(self, value):
        return value in self.dict

    def remove(self, value):
        del self.dict[value]
```

可以像下面这样来用上面的函数：

```
s = Set([1,2,3])
s.add(4)
print s.contains(4)       # True
s.remove(3)
print s.contains(3)       # False
```

2.2.7　函数式工具

在传递函数的时候，有时我们可能想部分地应用（或 curry）函数来创建新函数。下面是一个简单的例子，假设我们有一个含两个变量的函数：

```
def exp(base, power):
    return base ** power
```

我们想用它来创建一个单变量的函数 two_to_the。它的输入是一个幂次（power），输出的是 exp(2, power) 的结果。

当然，我们可以用 def 来实现，但它有时候使用起来并不太方便：

```
def two_to_the(power):
    return exp(2, power)
```

一个另辟蹊径的方法是使用 functools.partial：

```
from functools import partial
two_to_the = partial(exp, 2)      # 现在是一个包含一个变量的函数
print two_to_the(3)               # 8
```

如果你为后面的参数指定了名字，也能用 partial 来填充这些参数：

```
square_of = partial(exp, power=2)
print square_of(3)                  # 9
```

如果你 curry 中间函数的参数，就会变得混乱起来，所以要努力避免这么做。

偶尔我们也会使用函数 map、reduce 和 filter，它们为列表解析提供了函数式替换方案：

```
def double(x):
    return 2 * x

xs = [1, 2, 3, 4]
twice_xs = [double(x) for x in xs]        # [2, 4, 6, 8]
twice_xs = map(double, xs)                # 和上面一样
list_doubler = partial(map, double)       # double了一个列表的*function*
twice_xs = list_doubler(xs)               # 同样是[2, 4, 6, 8]
```

如果你提供了多个列表，可以对带有多个参数的函数使用 map：

```
def multiply(x, y): return x * y

products = map(multiply, [1, 2], [4, 5]) # [1 * 4, 2 * 5] = [4, 10]
```

类似地，filter 做了列表解析中 if 的工作：

```
def is_even(x):
    """True if x is even, False if x is odd"""
    return x % 2 == 0

x_evens = [x for x in xs if is_even(x)]      # [2, 4]
x_evens = filter(is_even, xs)                # 和上面一样
list_evener = partial(filter, is_even)       # filter了一个列表的*function*
x_evens = list_evener(xs)                    # 同样是[2, 4]
```

reduce 结合了列表的头两个元素，它们的结果又结合了列表的第 3 个元素，这个结果之后又结合了第 4 个元素，依次下去，直到得到一个单独的结果：

```
x_product = reduce(multiply, xs)           # = 1 * 2 * 3 * 4 = 24
list_product = partial(reduce, multiply)   # reduce了一个列表的*function*
x_product = list_product(xs)               # 同样是24
```

2.2.8 枚举

有时候，你可能想在一个列表上迭代，并且同时使用它的元素和元素的索引：

```
# 非Python用法
for i in range(len(documents)):
    document = documents[i]
    do_something(i, document)

# 也非Python用法
i = 0
for document in documents:
    do_something(i, document)
    i += 1
```

Python 惯用的解决方案是使用枚举（`enumerate`），它会产生 (`index, element`) 元组：

```
for i, document in enumerate(documents):
    do_something(i, document)
```

类似地，如果你只想要索引，则执行：

```
for i in range(len(documents)): do_something(i)     # 非Python用法
for i, _ in enumerate(documents): do_something(i)   # Python用法
```

我们会频繁用到枚举。

2.2.9 压缩和参数拆分

如果想把两个或多个列表压缩在一起，可以使用 `zip` 把多个列表转换为一个对应元素的元组的单个列表中：

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zip(list1, list2)        # 是[('a', 1), ('b', 2), ('c', 3)]
```

如果列表的长度各异，`zip` 会在第一个列表结束时停止。

可以使用一种特殊的方法“解压”一个列表：

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

其中的星号执行参数拆分（argument unpacking）。参数拆分使用 `pairs` 的元素作为独立的参数传给 `zip`。这就和调用以下函数的结果是一样的：

```
zip(('a', 1), ('b', 2), ('c', 3))
```

它返回 [('a','b','c'), ('1','2','3')]。

你可以在任何函数上使用参数拆分：

```
def add(a, b): return a + b

add(1, 2)     # 返回3
add([1, 2])   # TypeError!
add(*[1, 2])  # 返回3
```

参数拆分并不是特别有用，但当我们用到它的时候，会觉得这是个不错的技巧。

2.2.10 args和kwargs

假如我们想创建一个更高阶的函数，把某个函数 f 作为输入，并返回一个对任意输入都返回 f 值两倍的新函数：

```
def doubler(f):
    def g(x):
        return 2 * f(x)
    return g
```

这个函数在有些情况下可以实现：

```
def f1(x):
    return x + 1

g = doubler(f1)
print g(3)          # 8 (== ( 3 + 1) * 2)
print g(-1)         # 0 (== (-1 + 1) * 2)
```

但对于有多个参数的函数来说，就不适用：

```
def f2(x, y):
    return x + y

g = doubler(f2)
print g(1, 2)    # TypeError: g()只能有一个参数(给定了两个)
```

我们所需要的是一种指定一个可以取任意参数的函数的方法，利用参数拆分和一点点魔法就可以做到这一点：

```
def magic(*args, **kwargs):
    print "unnamed args:", args
    print "keyword args:", kwargs

magic(1, 2, key="word", key2="word2")
```

```
# 输出
# 未命名args: (1, 2)
# 关键词args: {'key2': 'word2', 'key': 'word'}
```

也就是说，当我们定义了这样一个函数时，args 是一个它的未命名参数的元组，而 kwargs 是一个它的已命名参数的 dict。反过来也适用，你可以使用一个 list（或者 tuple）和 dict 来给函数提供参数：

```
def other_way_magic(x, y, z):
    return x + y + z

x_y_list = [1, 2]
z_dict = { "z" : 3 }
print other_way_magic(*x_y_list, **z_dict)    # 6
```

参照以上例子，你可以充分利用这种技巧，随意发挥。我们将会只用它来创建可以将任意参数作为输入的高阶函数：

```
def doubler_correct(f):
    """works no matter what kind of inputs f expects"""
    def g(*args, **kwargs):
        """whatever arguments g is supplied, pass them through to f"""
        return 2 * f(*args, **kwargs)
    return g

g = doubler_correct(f2)
print g(1, 2) # 6
```

2.2.11　欢迎来到DataSciencester

新员工的入职培训到此结束。哦，当然，不要浪费任何所学的东西。

2.3　延伸学习

- Python 的教程比比皆是。官方的教程（https://docs.python.org/2/tutorial/）是不错的入门选择。
- 官方的 IPython 教程（http://ipython.org/ipython-doc/2/interactive/tutorial.html）不太理想，但其教学视频和报告（http://ipython.org/videos.html）还不错。此外，Wes Mckinney 的 *Python for Data Analysis*（O'Reilly，http://shop.oreilly.com/product/0636920023784.do）有一章非常好地讲解了 IPython。

第 3 章

可视化数据

我相信可视化是实现个人目标的最有力的手段之一!

——哈维 · 麦凯

数据可视化是数据科学家工具箱中的一个重要部分。创建可视化很容易，但创建优秀的可视化却很难。

数据可视化有两种主要用途：

- 探索数据
- 交流数据

本章我们将集中探讨你在探索数据和创建可视化过程中所用到的各项技能，可视化将贯穿本书剩余部分。就像本书大部分章节涉及的主题一样，数据可视化是一个非常丰富的研究领域，值得著书阐述。尽管如此，我们还是尝试着告诉你怎样辨别可视化的好坏。

3.1 matplotlib

有许多工具可以用来可视化数据，我们将使用的是应用最广的 `matplotlib` 库（尽管这暴露了它的年龄。详见 http://matplotlib.org/）。如果你的兴趣是制作用于网络的精良的交互可视化，它可能不是好的选择，但对于条形图、线图和散点图这些简单的图形来说，它很好用。

特别地，我们会使用 `matplotlib.pyplot` 模块。在最简单的应用中，`pyplot` 保持着一种内部状态，你可以在其中一步步地创建可视化。一旦创建工作完成，就可以保存（用

savefig()）或显示（用 show()）你的图形。

例如，制作一个（就像图 3-1 这样）非常简单的图形，就可以采取以下步骤：

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# 创建一幅线图,x轴是年份,y轴是gdp
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# 添加一个标题
plt.title("名义GDP")

# 给y轴加标记
plt.ylabel("十亿美元")
plt.show()
```

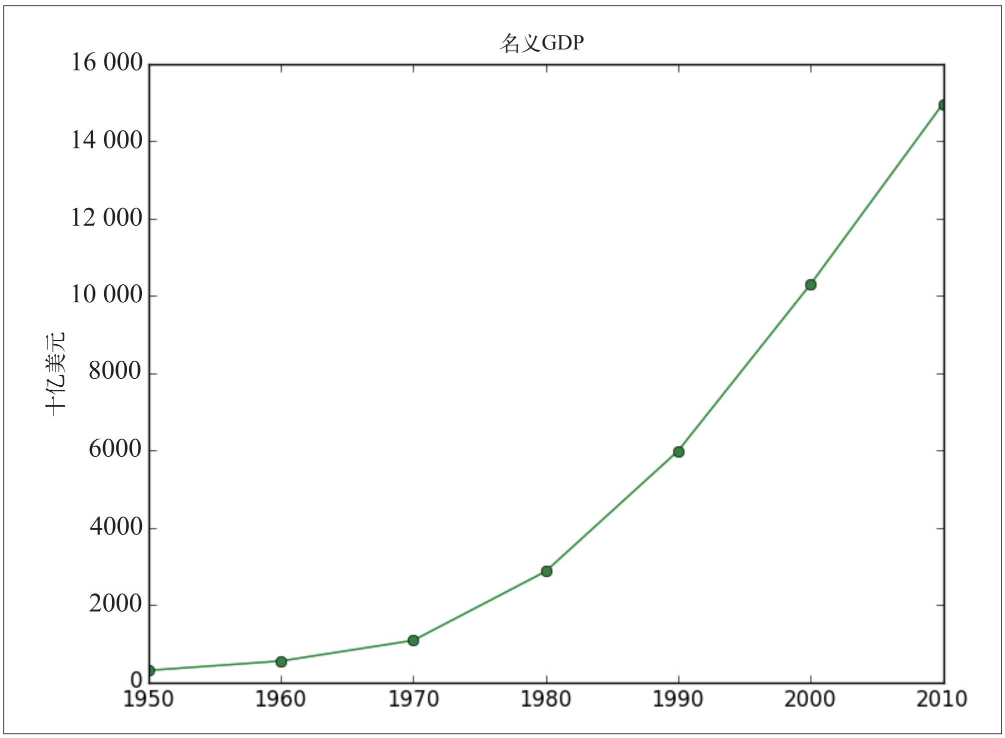


图 3-1：一个简单的线图

制作出版级别的精良图片要更加复杂，本章不作深入探讨。对图形进行自定义的方式有多种，如坐标轴标记、线型以及点的形状等。我们并不会面面俱到地讲解这些参数，只会在我们的例子中提到（并关注）其中的一些。

> 尽管我们并不会用到 matplotlib 太多的功能，但它着实能够制作出复杂的图中图、精致的图形格式和交互图形。如果你想做得比本书更深入，请查阅它的文档。

3.2　条形图

如果你想展示某些离散的项目集合中的数量是如何变化的，可以使用条形图。比如，图 3-2 显示了几部电影所获得的奥斯卡金像奖的数目：

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# 条形的默认宽度是0.8,因此我们对左侧坐标加上0.1
# 这样每个条形就被放置在中心了
xs = [i + 0.1 for i, _ in enumerate(movies)]

# 使用左侧x坐标[xs]和高度[num_oscars]画条形图
plt.bar(xs, num_oscars)

plt.ylabel("所获奥斯卡金像奖数量")
plt.title("我最喜爱的电影")

# 使用电影的名字标记x轴,位置在x轴上条形的中心
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)

plt.show()
```

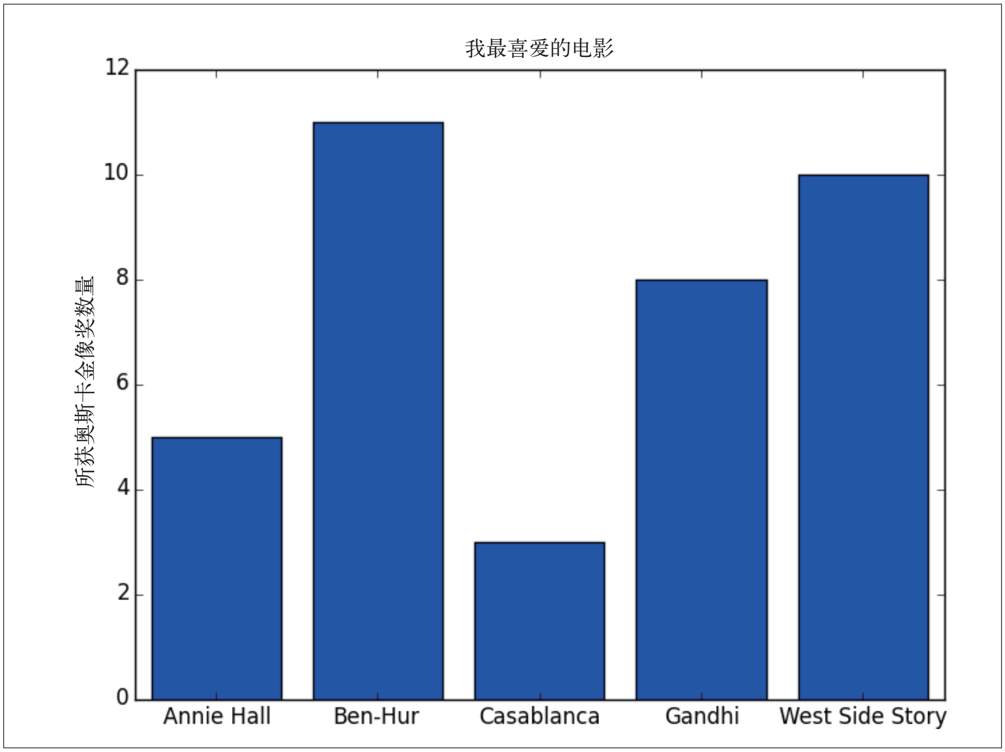


图 3-2：一个简单的条形图

条形图也可以用来绘制拥有大量数值取值的变量直方图，以此来探索这些取值是如何分布的，如图 3-3 所示。

```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # 每个条形向左侧移动4个单位
        histogram.values(),                # 给每个条形设置正确的高度
        8)                                 # 每个条形的宽度设置为8

plt.axis([-5, 105, 0, 5])                  # x轴取值从-5到105
                                           # y轴取值0到5

plt.xticks([10 * i for i in range(11)])    # x轴标记为0,10,...,100
plt.xlabel("十分相")
plt.ylabel("学生数")
plt.title("考试分数分布图")
plt.show()
```

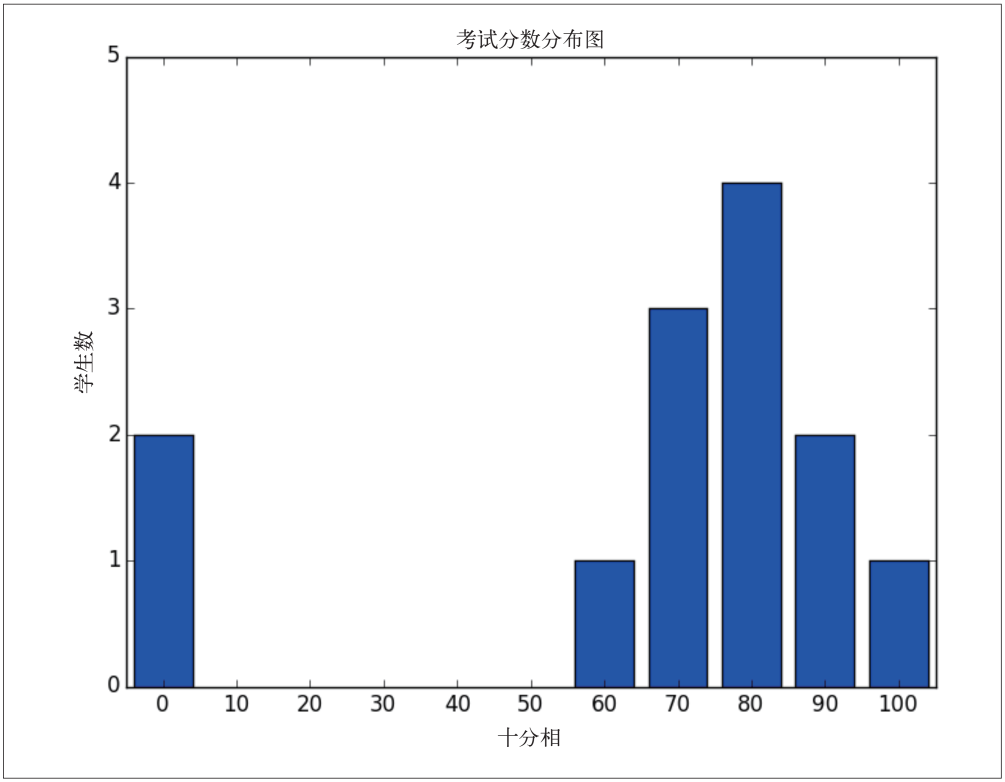


图 3-3：为直方图使用条形图

plt.bar 的第三个参数指定了条形的宽度，在这里我们选择宽度为 8（这样就在各个条形之间留出了小的间隔，因为 x 轴是以刻度 10 做标记的），而且把每个条形向左移了 4 个宽度。这样一来，（比如说）“80”这个条形的左边在 76，而右边在 84，因此它的中心在 80。

对 plt.axis 的调用表明我们希望 x 轴的范围是 −5~105（以使“0”到“100”这些条形可以完全显示），并且 y 轴的范围应限定在 0~5 之间。对 plt.xticks 的调用把 x 轴的刻度放在 0、10、20、……、100 这些位置。

在使用 plt.axis() 时要谨慎。在创建条形图时，y 轴不从 0 开始是一种特别不好的形式，因为这很容易误导人（见图 3-4）：

```
mentions = [500, 505]
years = [2013, 2014]

plt.bar([2012.6, 2013.6], mentions, 0.8)
plt.xticks(years)
plt.ylabel("听到有人提及‘数据科学’的次数")

# 如果不这么做,matplotlib会把x轴的刻度标记为0和1
```

```
# 然后会在角上加上+2.013e3(糟糕的matplotlib操作！)
plt.ticklabel_format(useOffset=False)

# 这会误导y轴只显示500以上的部分
plt.axis([2012.5,2014.5,499,506])
plt.title("快看如此'巨大'的增长！")
plt.show()
```

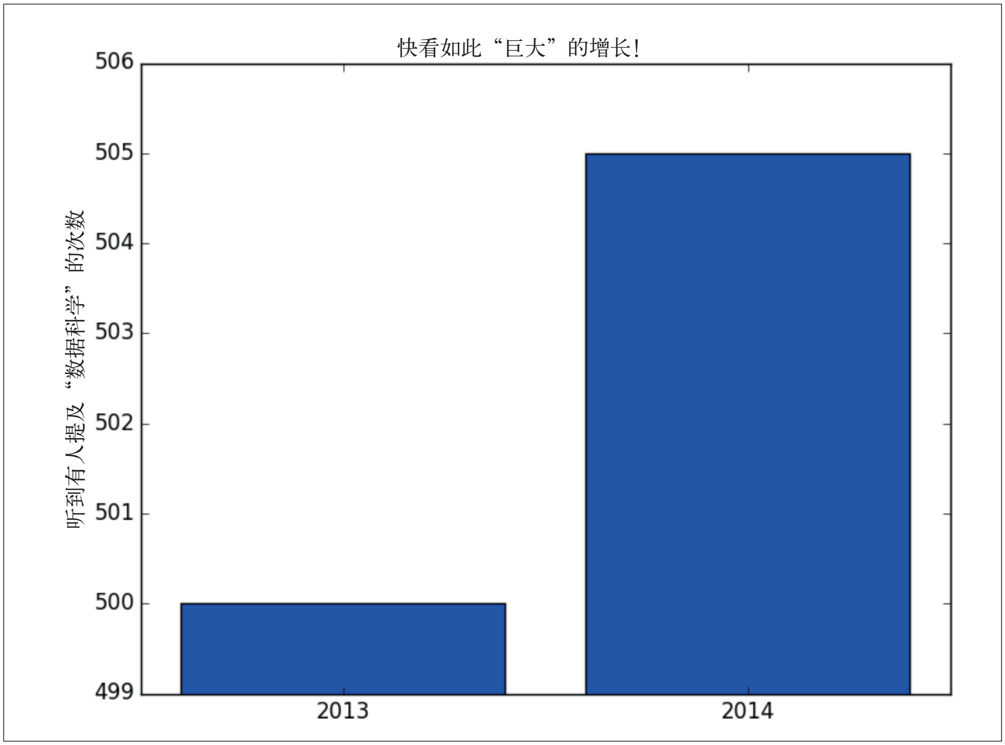


图 3-4：一个有误导性 *y* 轴的条形图

在图 3-5 中，我们使用了一种更合理的轴，这样它看起来就不那么异常了：

```
plt.axis([2012.5,2014.5,0,550])
plt.title("增长不那么巨大了")
plt.show()
```

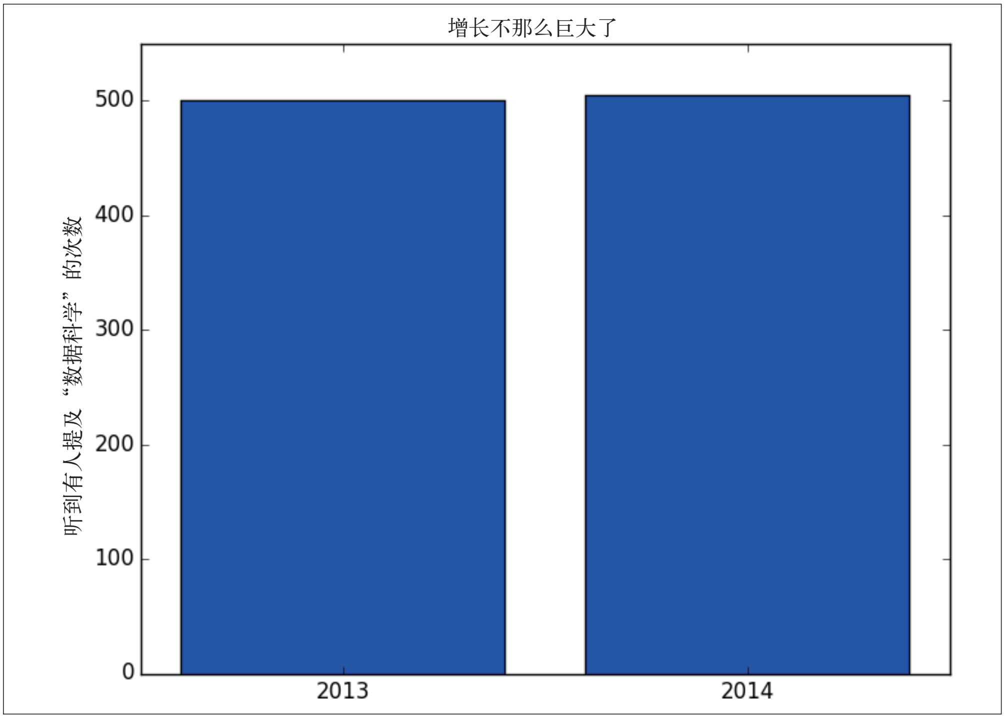


图 3-5：*y* 轴正常的同一个条形图

3.3 线图

前面提过，可以用 plt.plot() 来制作线图。这种图形可以用来清晰地显示某种事物的趋势，如图 3-6 所示：

```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# 可以多次调用plt.plot
# 以便在同一个图上显示多个序列
plt.plot(xs, variance,     'g-',  label='variance')     # 绿色实线
plt.plot(xs, bias_squared, 'r-.', label='bias^2')       # 红色点虚线
plt.plot(xs, total_error,  'b:',  label='total error')  # 蓝色点线

# 因为已经对每个序列都指派了标记
# 所以可以自由地布置图例
# loc=9指的是“顶部中央”
plt.legend(loc=9)
plt.xlabel("模型复杂度")
plt.title("偏差-方差权衡图")
plt.show()
```

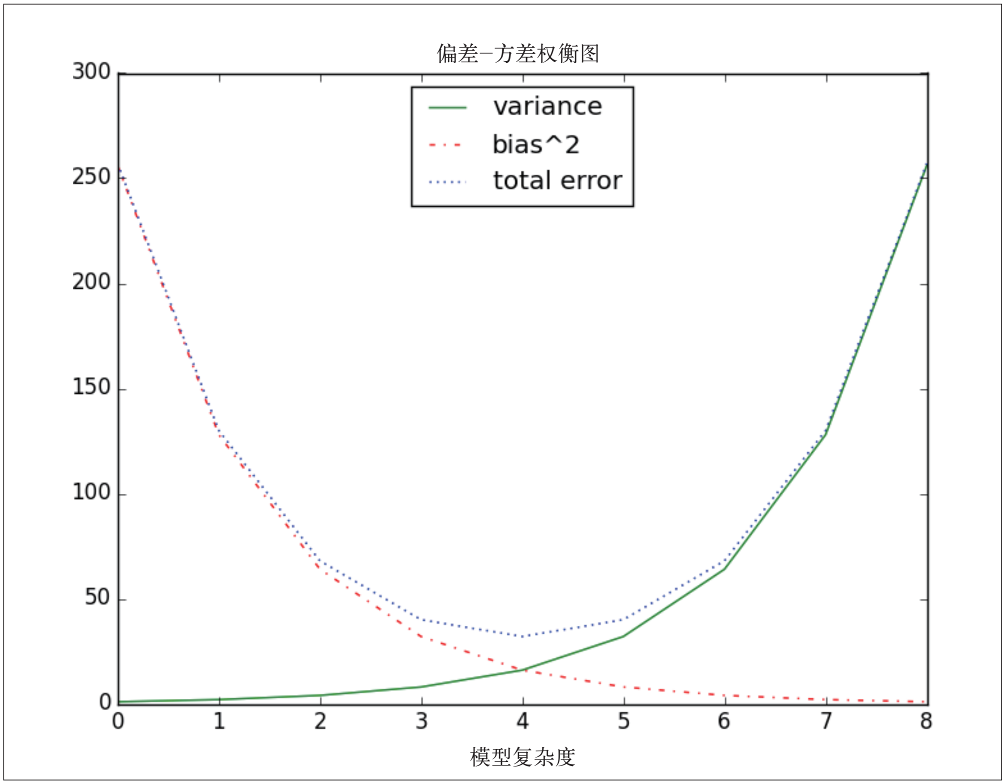


图 3-6：带有图例的几个线图

3.4 散点图

散点图是显示成对数据集的可视化关系的好选择。比如，图 3-7 显示了你的用户们已有的朋友数和他们每天花在网站上的分钟数之间的关系：

```
friends = [ 70,  65,  72,  63,  71,  64,  60,  64,  67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels =  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

plt.scatter(friends, minutes)

# 每个点加标记
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
        xy=(friend_count, minute_count), # 把标记放在对应的点上
        xytext=(5, -5),                  # 但要有轻微偏离
        textcoords='offset points')

plt.title("日分钟数与朋友数")
plt.xlabel("朋友数")
```

```
plt.ylabel("花在网站上的日分钟数")
plt.show()
```

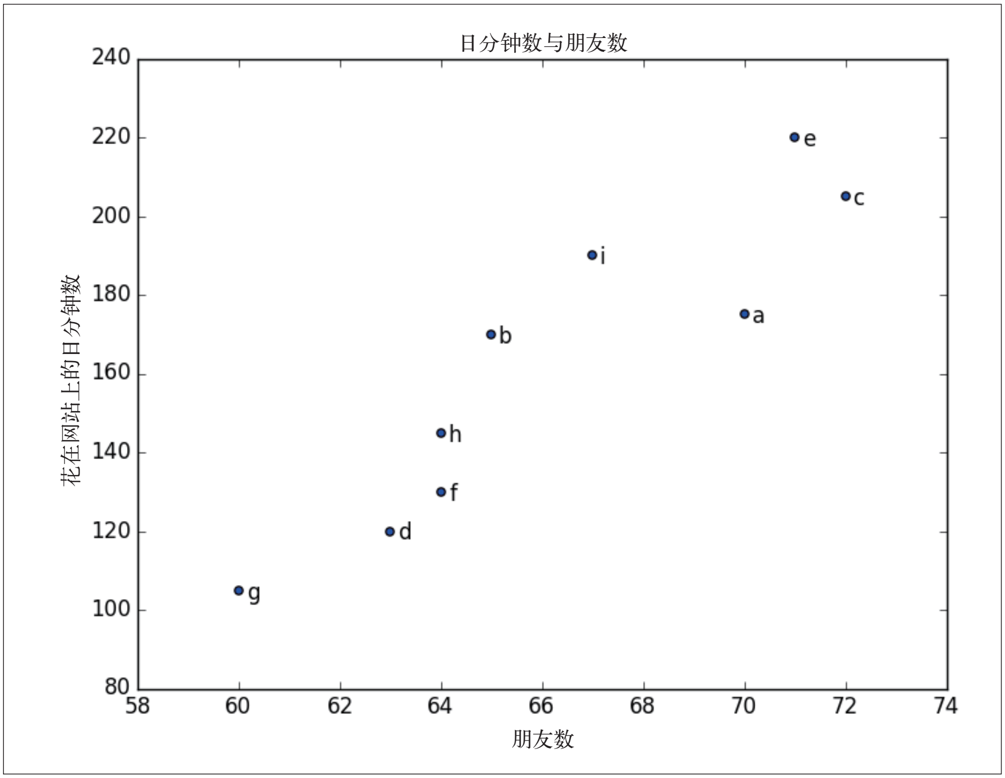


图 3-7：朋友数与花在网站上的分钟数之间的关系散点图

当你分散了可比较的变量，如果让 `matplotlib` 选择刻度，可能会得到一个误导性的图，如图 3-8 所示：

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren't Comparable")
plt.xlabel("测验1的分数")
plt.ylabel("测验2的分数")
plt.show()
```

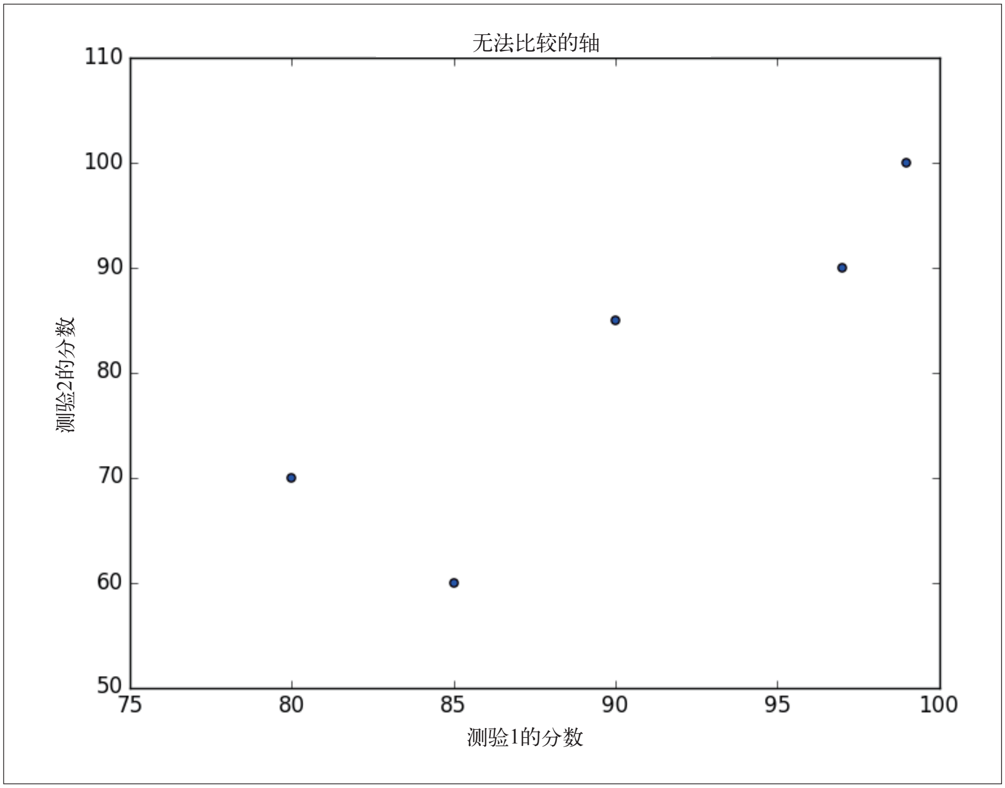


图 3-8：带有无法比较的轴的散点图

如果我们引入对 `plt.axis("equal")` 的调用，图形（图 3-9）会更精确地显示大多数变化是发生在测验 2 上的。

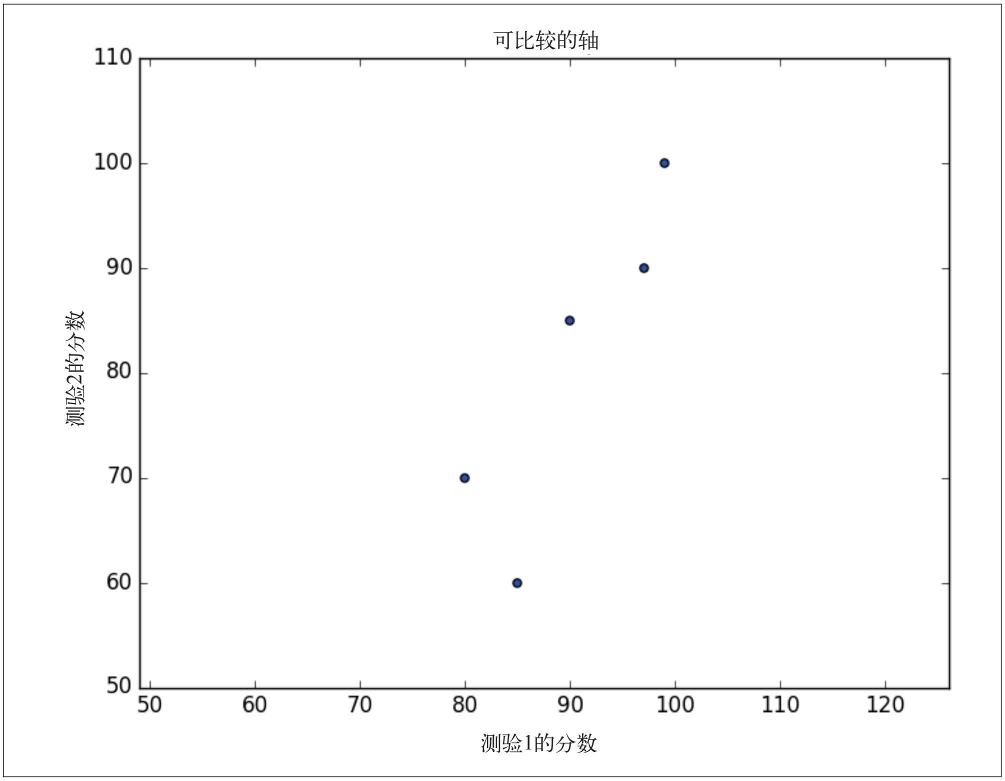


图 3-9：带有可比较的轴的同一个散点图

以上这些内容对于开始进行可视化已经足够了，本书后面还会讲解更多关于可视化的知识。

3.5 延伸学习

- seaborn（http://web.stanford.edu/~mwaskom/software/seaborn/）基于 `matplotlib` 构建，让你可以轻松地制作更漂亮也更复杂的可视化。
- D3.js（http://d3js.org/）是一个 JavaScript 库，可以制作精致的基于网络的交互可视化。尽管它并不存在于 Python 中，但它很时髦，并且应用广泛，值得你花时间来了解它。
- Bokeh（http://bokeh.pydata.org/en/latest/）是一个较新的库，它将 D3 风格的可视化引入了 Python 中。
- ggplot（https://pypi.python.org/pypi/ggplot）是流行的 R 库 `ggplot2` 的 Python 接口。`ggplot2` 被广泛用于生成“出版级别”的图形图像。如果你是一个热情的 `ggplot2` 用户，它可能是非常有趣的工具，但如果你不熟悉它，可能会觉得使用起来有些困难。

第 4 章
线性代数

百无一用是代数。

——比利·康诺利

线性代数是数学的一个分支，研究向量空间。我不指望用一章就能教会你线性代数，但这章涵盖了数据科学的大量概念和技术，因此你至少试着学习一下。本章所学内容在本书后续部分会大量应用。

4.1 向量

抽象地说，向量是指可以加总（以生成新的向量），可以乘以标量（即数字），也可以生成新的向量的对象。

具体来说（对我们而言），向量是有限维空间的点。即使你本无意视你的数据为向量，将数值数据表示为向量也是非常好的处理方式。

比如，如果你有很多人的身高、体重、年龄数据，就可以把数据记为三维向量（`height, weight, age`）。如果你教的一个班有四门考试，就可以把学生成绩记为四维向量（`exam1, exam2, exam3, exam4`）。

最简单的入门方法是将向量表示为数字的列表。一个包含三个数字的列表对应一个三维空间的向量，反之亦然：

```
height_weight_age = [70,  # 英寸
                     170, # 磅
                     40 ] # 岁

grades = [95,  # 考试1
          80,  # 考试2
          75,  # 考试3
          62 ] # 考试4
```

这种方式的一个问题在于向量算法的应用。由于 Python 中的列表不同于向量（因此无法直接对向量运算），我们需要自己提前构建相应算法工具。现在就开始构建吧！

首先，我们常常需要对两个向量做加法。向量以*分量方式*（componentwise）做运算。这意味着，如果两个向量 `v` 和 `w` 长度相同，那它们的和就是一个新的向量，其中向量的第一个元素等于 `v[0] + w[0]`，第二个元素等于 `v[1] + w[1]`，以此类推。（如果两个向量长度不同，则不能相加。）

例如，向量 `[1, 2]` 加上向量 `[2, 1]` 等于 `[1 + 2, 2 + 1]` 或 `[3, 3]`，如图 4-1 所示。

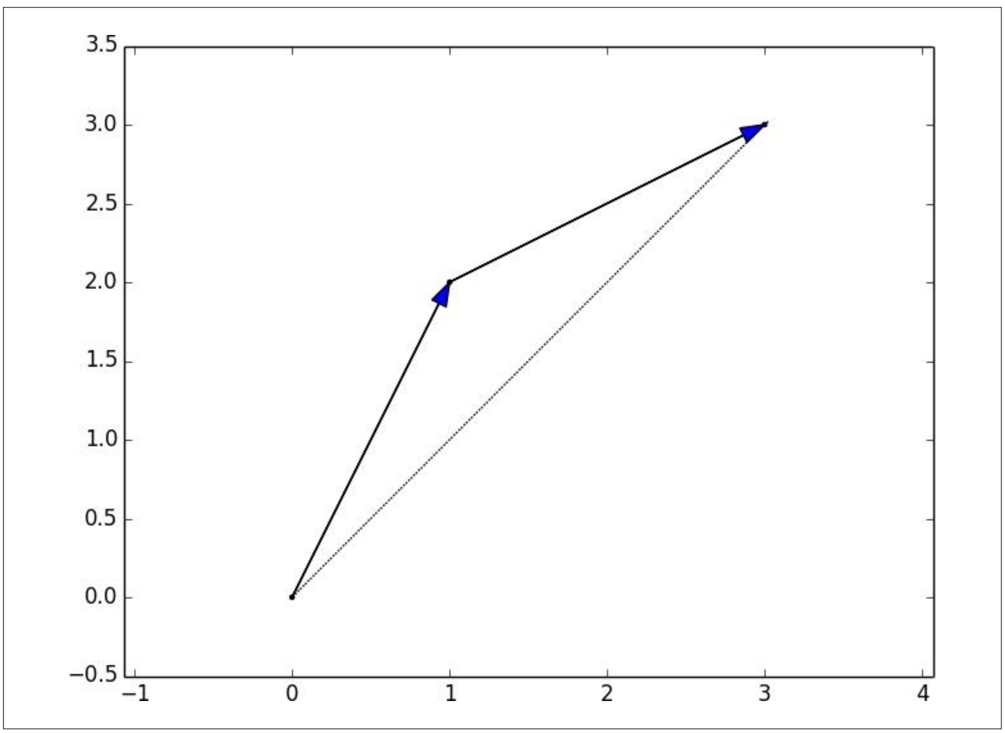


图 4-1：两个向量相加

我们可以很容易地实现这个功能：对向量调用 `zip` 函数，同时用列表解析使向量的相应元素相加：

```
def vector_add(v, w):
    """adds corresponding elements"""
    return [v_i + w_i
            for v_i, w_i in zip(v, w)]
```

同样，对两个向量做减法，只需要使向量的相应元素相减：

```
def vector_subtract(v, w):
    """subtracts corresponding elements"""
    return [v_i - w_i
            for v_i, w_i in zip(v, w)]
```

有时，我们需要对一系列向量做加法。即生成一个新向量，其第一个元素是这一系列向量第一个元素的和，第二个元素是这一系列向量第二个元素的和，以此类推。最简单的方法是每次递加一个向量：

```
def vector_sum(vectors):
    """sums all corresponding elements"""
    result = vectors[0]                         # 从第一个向量开始
    for vector in vectors[1:]:                  # 之后遍历其他向量
        result = vector_add(result, vector)     # 最后计入总和
    return result
```

当你思考这个解决方法时，我们正通过 vector_add 函数，即使用 reduce 的方式来加总这一系列的向量。换句话说，我们用高级的函数更加简洁地实现了这个功能：

```
def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

或者：

```
vector_sum = partial(reduce, vector_add)
```

这最后一种方法很简洁、巧妙，但可能相比之下没有前几种有用处。

当然，我们有时也需要给一个向量乘以一个标量，这时只需将向量的每个元素乘以那个数字：

```
def scalar_multiply(c, v):
    """c is a number, v is a vector"""
    return [c * v_i for v_i in v]
```

我们也可以计算一系列向量（长度相同）的均值：

```
def vector_mean(vectors):
    """compute the vector whose ith element is the mean of the
    ith elements of the input vectors"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))
```

一个不常见的功能是点乘（dot product）。两个向量的点乘表示对应元素的分量乘积之和：

```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i
               for v_i, w_i in zip(v, w))
```

点乘衡量了向量 v 在向量 w 方向延伸的程度。例如，如果 `w=[1, 0]`，则 `dot(v, w)` 就是 `v` 的第一个元素。点乘的另一个解释是将 v 在 w 上投影所得到的向量的长度（如图 4-2）：

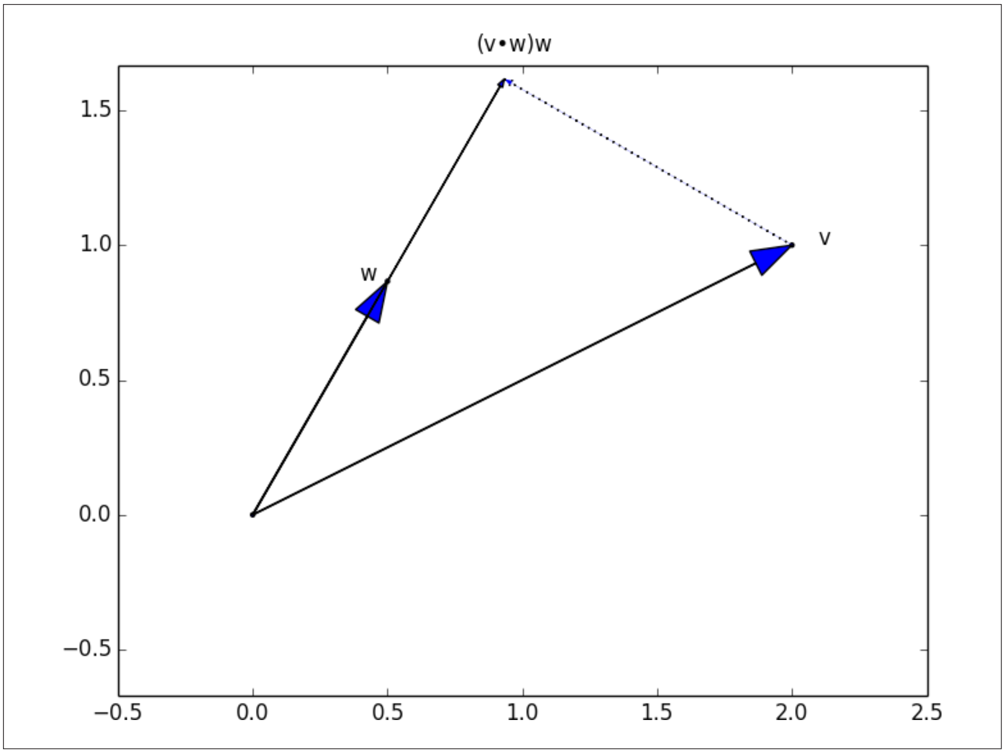


图 4-2：点乘即向量投影

通过点乘很容易计算一个向量的平方和：

```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
```

可以用来计算向量的大小（或长度）：

```
import math

def magnitude(v):
    return math.sqrt(sum_of_squares(v))    # math.sqrt是平方根函数
```

现在，我们得到了为计算两个向量的距离所需要的所有部分，定义如下：

$$\sqrt{(v_1 - w_1)^2 + \cdots + (v_n - w_n)^2}$$

```
def squared_distance(v, w):
    """(v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(vector_subtract(v, w))

def distance(v, w):
   return math.sqrt(squared_distance(v, w))
```

写成下式（与上式等价）更清晰：

```
def distance(v, w):
    return magnitude(vector_subtract(v, w))
```

有了这些关于向量的概念和计算，我们就可以开始探讨数据科学了。本书后续部分会大量使用这些概念。

> 用列表来表示向量很有利于概念阐释，但对性能却影响很糟。
>
> 实际编程中，你需要使用 NumPy 库。这个库中有包含各种算法操作的高性能数组类。

4.2 矩阵

*矩阵*是一个二维的数据集合。我们将矩阵表示为列表的列表，每个内部列表的大小都一样，表示矩阵的一行。如果 A 是一个矩阵，那么 A[i][j] 就表示第 i 行第 j 列的元素。按照数学表达的惯例，我们通常用大写字母表示矩阵。例如：

```
A = [[1, 2, 3],   # A有2行3列
     [4, 5, 6]]

B = [[1, 2],      # B有3行2列
     [3, 4],
     [5, 6]]
```

> 在数学中，矩阵的第一行通常称为“第 1 行”，第一列通常称为“第 1 列”。而我们需要将矩阵的形式和 Python 的列表统一起来：Python 中的列表从 0 开始索引，所以，我们将矩阵的第一行称为“第 0 行”，将第一列称为“第 0 列”。

基于列表的列表这种表达形式，矩阵 A 具有 len(A) 行和 len(A[0]) 列，我们把这称作它的形状（shape）：

```
def shape(A):
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0   # 第一行中元素的个数
    return num_rows, num_cols
```

如果一个矩阵有 n 行 k 列，则可以记为 $n\times k$ 矩阵。我们可以把这个 $n\times k$ 矩阵的每一行都当作一个长度为 k 的向量，把每一列都当作一个长度为 n 的向量：

```
def get_row(A, i):
    return A[i]             # A[i]是第i行

def get_column(A, j):
    return [A_i[j]          # 第A_i行的第j个元素
            for A_i in A]   # 对每个A_i行
```

同样，我们也可以根据形状和用来生成元素的函数来创建矩阵。可以通过一个嵌套的列表解析来实现：

```
def make_matrix(num_rows, num_cols, entry_fn):
    """returns a num_rows x num_cols matrix
    whose (i,j)th entry is entry_fn(i, j)"""
    return [[entry_fn(i, j)             # 根据i创建一个列表
             for j in range(num_cols)]  # [entry_fn(i, 0), ... ]
            for i in range(num_rows)]   # 为每一个i创建一个列表
```

有了这个函数，就可以生成一个 5×5 的单位矩阵（对角线元素是 1，其他元素是 0）：

```
def is_diagonal(i, j):
    """1's on the 'diagonal', 0's everywhere else"""
    return 1 if i == j else 0

identity_matrix = make_matrix(5, 5, is_diagonal)

# [[1, 0, 0, 0, 0],
#  [0, 1, 0, 0, 0],
#  [0, 0, 1, 0, 0],
#  [0, 0, 0, 1, 0],
#  [0, 0, 0, 0, 1]]
```

矩阵的重要性不言而喻，原因有以下几个。

首先，可以通过将每个向量看成是矩阵的一行，来用矩阵表示一个包含多维向量的数据集。例如，如果有 1000 个人的身高、体重和年龄，就可以创建一个 1000×3 的矩阵：

```
data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ....
       ]
```

第二，随后我们会看到，可以用一个 $n\times k$ 矩阵表示一个线性函数，这个函数将一个 k 维的

向量映射到一个 n 维的向量上。我们使用的很多技术与概念都隐含着这样的函数。

第三，可以用矩阵表示二维关系。在第 1 章中，我们曾经将网络边际表示为数据对 (i, j) 的集合。我们还可以通过建立矩阵 A 来实现这个描述：如果节点 i 和节点 j 有关系，则用矩阵的元素 A[i][j] 为 1 来表示；若节点 i 和节点 j 没有关系，则用 A[i][j] 为 0 来表示。

回想前面讲过的关系：

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

可以通过矩阵形式再现：

```
     #     用户 0 1 2 3 4 5 6 7 8 9
     #
friendships = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # 用户 0
               [1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # 用户 1
               [1, 1, 0, 1, 0, 0, 0, 0, 0, 0], # 用户 2
               [0, 1, 1, 0, 1, 0, 0, 0, 0, 0], # 用户 3
               [0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # 用户 4
               [0, 0, 0, 0, 1, 0, 1, 1, 0, 0], # 用户 5
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # 用户 6
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # 用户 7
               [0, 0, 0, 0, 0, 0, 1, 1, 0, 1], # 用户 8
               [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]] # 用户 9
```

如果关系很少，这种表示形式的效率就会很低，因为必须存储很多零。但是，通过矩阵表示可以快速地查找确认某两个节点是否是连接的——通过一个矩阵查找函数即可，不必（有可能会）搜索每条边：

```
friendships[0][2] == 1  # true,0和2是朋友
friendships[0][8] == 1  # false,0和8不是朋友
```

同样，若想找到一个节点的所有连接，只需检查这个节点所在的列（或行）：

```
friends_of_five = [i                                            # 仅需在
                   for i, is_friend in enumerate(friendships[5])   # 一行中
                   if is_friend]                                # 查找
```

为了加速这个搜寻过程，我们之前给每个节点对象都添加了一个连接列表。但对于太大的、扩展性强的图形来说，成本太高，维护也很难。

本书随后还会探讨这些矩阵。

4.3 延伸学习

- 线性代数在数据科学家中应用很广（通常是隐式使用，且不谙此道的人也经常使用）。阅读这方面的教材是个好主意。网上也有很多免费资源，如：

  - UC Davis 提供的有关线性代数的资源（https://www.math.ucdavis.edu/~linear/）；
  - 圣迈克尔大学提供的有关线性代数的资源（http://joshua.smcvt.edu/linearalgebra/）；
  - 对喜欢挑战的读者来说，*Linear Algebra Done Wrong*（http://www.math.brown.edu/~treil/papers/LADW/LADW.html）是一本进阶读物。
- 如果你使用 NumPy（http://www.numpy.org/）的话，我们此处创建的所有工具都可以免费使用（你赚到了）。

第5章

统计学

事实如磐石，统计似蒲草。

——马克 · 吐温

统计学是我们赖以理解数据的数学与技术。这是一个庞大繁荣的领域，要用图书馆的一整个书架（甚至一整间屋子）的书来阐述，绝非一本书的一个章节所能尽述，所以本书的探讨必然不会太深入。我在此抛砖引玉，讲述一些刚好能激发你的兴趣的内容，供你自行深入学习与开拓。

5.1　描述单个数据集

凭借口碑与运气，DataSciencester 已经发展了数十名成员。这时，融资部门的副总来问你要一些关于你的成员有多少朋友的描述，以此来确定他潜在的电梯演说对象。

运用第 1 章中学到的技术可以很容易地生成这个数据。但你现面临的问题是如何*描述*它。

对任何数据集，最简单的描述方法就是数据本身：

```
num_friends = [100, 49, 41, 40, 25,
               # ……等等许多
              ]
```

对足够小的数据集来说，这甚至可以说是最好的描述方法。但随着数据规模变大，这就显得笨拙又含混了。（想象一个包含一亿个数字的列表。）为此，我们使用统计来提炼和表达数据的相关特征。

首先，我们通过 Couner 和 plt.bar() 把你的朋友数绘成直方图（图 5-1）：

```
friend_counts = Counter(num_friends)
xs = range(101)                         # 最大值是100
ys = [friend_counts[x] for x in xs]     # height刚好是朋友的个数
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("朋友数的直方图")
plt.xlabel("朋友个数")
plt.ylabel("人数")
plt.show()
```

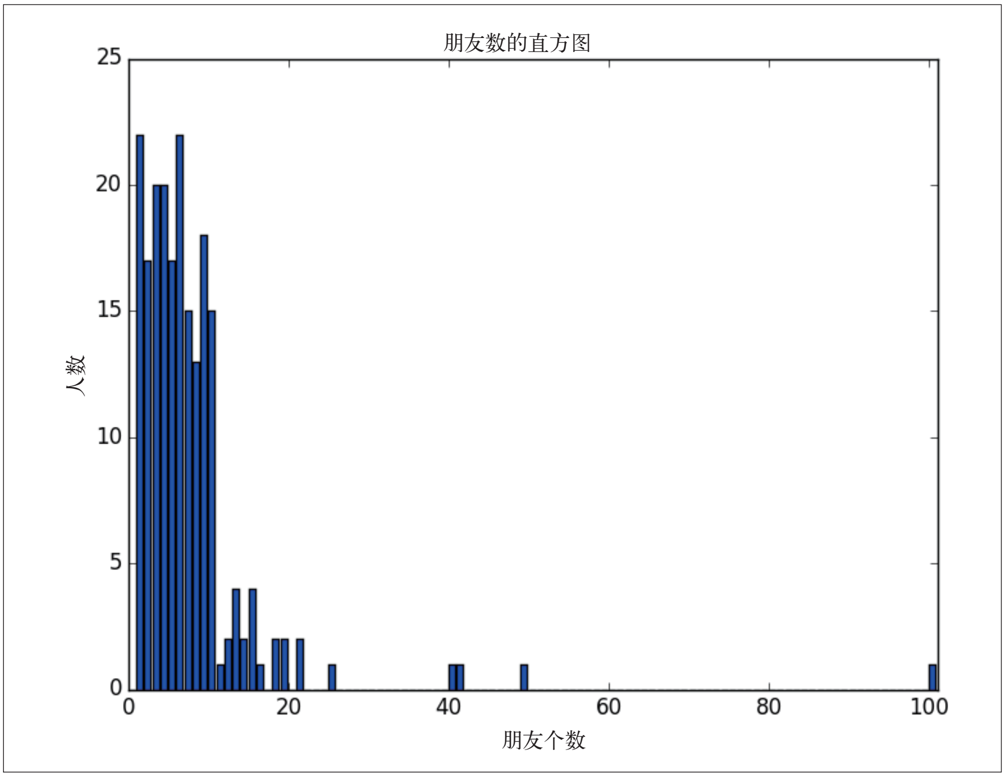


图 5-1：朋友数的直方图

不幸的是，这幅图难以用来进行交流，所以你需要再提炼一些统计量。数据点个数大概就是最简单的统计量了：

```
num_points = len(num_friends)               # 204
```

也许你会对数据集的最大值和最小值感兴趣：

```
largest_value = max(num_friends)            # 100
smallest_value = min(num_friends)           # 1
```

如果你想知道特定位置的值，可以这样做：

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]         # 1
second_smallest_value = sorted_values[1]  # 1
second_largest_value = sorted_values[-2]  # 49
```

当然，这仅仅是开始。

5.1.1 中心倾向

我们常常希望了解数据中心位置的一些概念。一个常用的方法是使用均值（mean 或 average），即用数据和除以数据个数：

```
# 如果没有从__future__导入division,那就是不对的
def mean(x):
    return sum(x) / len(x)

mean(num_friends)   # 7.333333
```

如果你有两个数据点，均值就意味着两点的中间点。随着数据集中点数的增加，均值点会移动，但它始终取决于每个点的取值。

邮 我们常常也会用到中位数（median），它是指数据中间点的值（如果数据点的个数是奇
电 数），或者中间两个点的平均值（如果数据点的个数是偶数）。

例如，如果在排序向量 x 上有五个数据点，那么中位数就是 x[5 // 2] 或 x[2]。如果有六个数据点，则中位数是 x[2]（第三个点）与 x[3]（第四个点）的平均数。

注意——和均值不同——中位数并不依赖于每一个数据的值。例如，即便数据集中最大的点变得更大（或最小的点变得更小），中间的数据点都不会变，意味着中位数也不会变。

median 函数很可能比你想象的更复杂一些，主要是因为数据集中数据个数奇偶性的不同：

```
def median(v):
    """finds the 'middle-most' value of v"""
    n = len(v)
    sorted_v = sorted(v)
    midpoint = n // 2

    if n % 2 == 1:
        # 如果是奇数,返回中间值
        return sorted_v[midpoint]
    else:
        # 如果是偶数,返回中间两个值的均值
        lo = midpoint - 1
        hi = midpoint
        return (sorted_v[lo] + sorted_v[hi]) / 2
        median(num_friends) # 6.0
```

很明显，均值的计算更简单，并且它会随着数据变化而平稳地变化。如果有 n 个数据点，其中某一个点的值增加了 e，则均值随之增加 e/n。（这使得均值适用于各种微分运算）但是为了计算中位数，得先对数据排序。并且，如果其中一个数据点的值增加了 e，那么中位数有可能也增加 e，有可能增加一个小于 e 的数，也有可能根本不变（这取决于其他的数据）。

> 事实上，不排序也可以使用一些生僻的技巧有效地算出中位数（https://en.wikipedia.org/wiki/Quickselect）。但这些技巧不但不易理解，而且超出了本书的讲解范围。所以我们先排序再计算。

同时，均值对数据中的异常值非常敏感。如果最具人缘的用户有 200 个朋友（不是 100），均值会上升至 7.82，而中位数不变。如果异常值属于不良数据（或者对我们试图理解的现象不具有代表性），那么均值会误导我们。举一个老生常谈的例子，20 世纪 80 年代，北卡罗来纳大学起薪最高的专业是地理学，因为球星迈克尔 · 乔丹曾就读于此，均值计算就包含了这个“异常值”。

中位数的一个泛化概念是*分位数*（quantile），它表示少于数据中特定百分比的一个值。（中位数表示少于 50% 的数据的一个值。）

```
def quantile(x, p):
    """returns the pth-percentile value in x"""
    p_index = int(p * len(x))
    return sorted(x)[p_index]

quantile(num_friends, 0.10) # 1
quantile(num_friends, 0.25) # 3
quantile(num_friends, 0.75) # 9
quantile(num_friends, 0.90) # 13
```

还有一个不太常用的概念*众数*（mode），它是指出现次数最多的一个或多个数：

```
def mode(x):
    """returns a list, might be more than one mode"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.iteritems()
            if count == max_count]

mode(num_friends)       # 1 和 6
```

但是，最常用的还是均值。

5.1.2　离散度

*离散度*是数据的离散程度的一种度量。通常，如果它所统计的值接近零，则表示数据聚集

在一起，*离散程度很小*，如果值很大（无论那意味着什么），则表示数据的*离散度很大*。例如，一个简单的度量是*极差*（range），指最大元素与最小元素的差：

```
# "range" 在Python中已经有特定的含义,所以我们换一个不同的名字
def data_range(x):
    return max(x) - min(x)

data_range(num_friends) # 99
```

极差恰好为零，意味着数据集中最大值和最小值相等，这种情形只有在 `x` 中的元素全部相同时才会发生，意味着数据没有离散。相反，如果极差很大，说明最大元素比最小元素大很多，数据离散度很高。

和中位数一样，极差也不真正依赖于整个数据集。一个只包含 0 和 100 的数据集，和一个包含 0、1 以及很多个 50 的数据集，两者的极差相同。但看起来第一个数据集的离散度“应该”更高。

离散度的另一个更复杂的度量是*方差*（variance），计算方式如下：

```
def de_mean(x):
    """translate x by subtracting its mean (so the result has mean 0)"""
    x_bar = mean(x)
    return [x_i - x_bar for x_i in x]

def variance(x):
    """assumes x has at least two elements"""
    n = len(x)
    deviations = de_mean(x)
    return sum_of_squares(deviations) / (n - 1)

variance(num_friends) # 81.54
```

这个概念看起来似乎是各个数值分别与其均值之差的平方的均值，但我们除以的是 `n-1` 而不是 `n`。事实上，如果样本取自更大的总体，`x_bar` 就是真实均值的估值，意味着 `(x_i - x_bar) ** 2` 是 `x_i` 的方差对均值的低估值，所以我们除以 `n-1` 而不是 `n`。更多信息请查看维基百科。

现在，无论我们的数据是什么单位（即“朋友”），所有中心倾向的度量都是同一单位。极差的单位也与此相同。但是，方差的单位是原数据单位的*平方*（即“平方朋友”）。然而，用方差很难给出直观的比较，所以我们更常使用*标准差*（standard deviation）：

```
def standard_deviation(x):
    return math.sqrt(variance(x))

standard_deviation(num_friends) # 9.03
```

极差和标准差也都有我们之前提到的均值计算常遇到的异常值问题。再看之前的例子，如果我们最具人缘的用户有 200 个朋友，标准差就变为 14.89，增加了 60%！

一种更加稳健的方案是计算 75% 的分位数和 25% 的分位数之差：

```
def interquartile_range(x):
    return quantile(x, 0.75) - quantile(x, 0.25)

interquartile_range(num_friends) # 6
```

相对来说，这种计算不易受到一小部分异常值的影响。

5.2 相关

DataSciencester 战略发展部的副总持有这样一种想法，即用户在某个网站上花费的时间与其在这个网站上拥有的朋友数相关（他并不是一个无所事事的领导）。现在，他要求你来验证这个想法。

通过分析研究流量日志，你设法做出了一个 daily_minutes 列表，这个列表描述了每个用户每天在 DataSciencester 花费了多长时间。你还对这个列表排了序，使它的元素和你之前的列表 num_friends 的元素对应了起来，以便进一步研究两个度量之间的关系。

我们先来看一下*协方差*（covariance），这个概念是方差的一个对应词。方差衡量了单个变量对均值的偏离程度，而协方差衡量了两个变量对均值的串联偏离程度：

```
def covariance(x, y):
    n = len(x)
    return dot(de_mean(x), de_mean(y)) / (n - 1)

covariance(num_friends, daily_minutes) # 22.43
```

回想一下点乘（dot）的概念，它意味着对应的元素对相乘后再求和。如果向量 x 和向量 y 的对应元素同时大于它们自身序列的均值，或者同时小于它们自身序列的均值，那将为求和贡献一个正值。如果其中一个元素大于自身的均值，而另一个小于自身的均值，那将为求和贡献一个负值。因此，如果协方差是一个大的正数，就意味着如果 y 很大，那么 x 也很大，或者如果 y 很小，那么 x 也很小。如果协方差为负而且绝对值很大，就意味着 x 和 y 一个很大，而另一个很小。接近零的协方差意味着以上关系都不存在。

但是，这个数字很难解释，原因如下。

- 它的单位是输入单位的乘积（即朋友 – 分钟 – 每天），难于理解。（“朋友 – 分钟 – 每天”是什么鬼？）
- 如果每个用户的朋友数增加到两倍（但分钟数不变），方差会增加至两倍。但从某种意义上讲，变量的相关度是一样的。换句话讲，很难说“大”的协方差意味着什么。

因此，相关是更常受到重视的概念，它是由协方差除以两个变量的标准差：

```
def correlation(x, y):
    stdev_x = standard_deviation(x)
    stdev_y = standard_deviation(y)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(x, y) / stdev_x / stdev_y
    else:
        return 0    # 如果没有变动,相关系数为零

correlation(num_friends, daily_minutes) # 0.25
```

相关系数没有单位，它的取值在 −1（完全反相关）和 1（完全相关）之间。相关值 0.25 代表一个相对较弱的正相关。

但是，我们忽略了对数据的检查。看图 5-2。

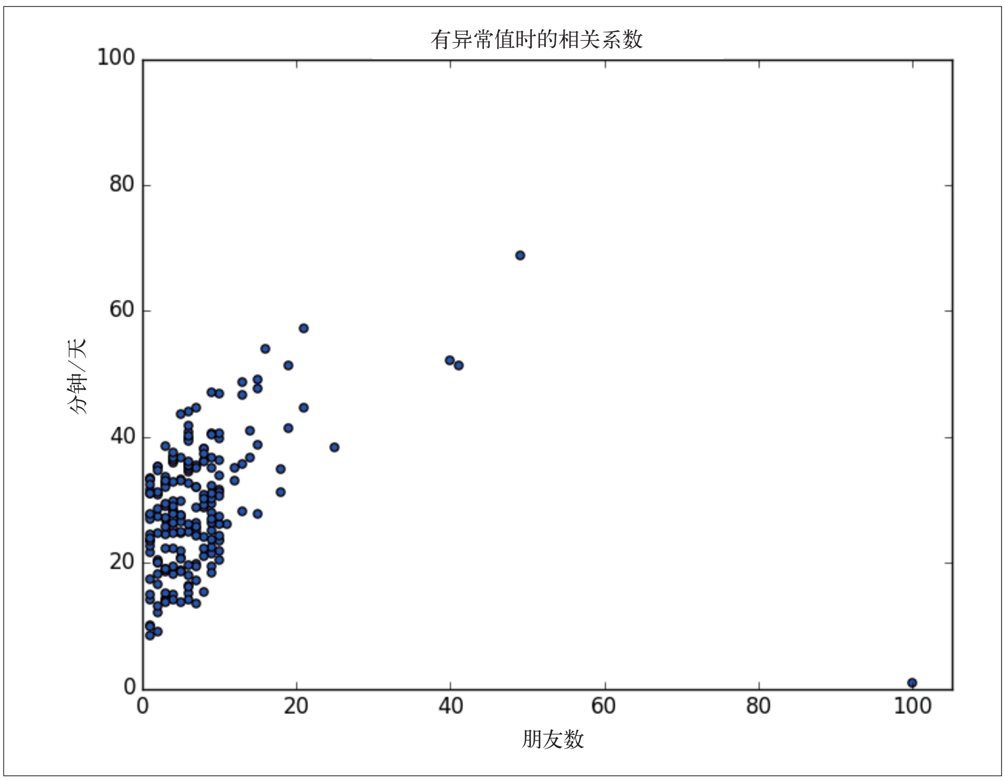


图 5-2：有异常值时的相关系数

图中那个有 100 个朋友的用户（每天只在网上花费 1 分钟）是一个明显的异常值，相关系数的计算对异常值非常敏感。如果我们计算时希望忽略这个人，该怎么做呢？如下所示：

```
outlier = num_friends.index(100) # outlier的索引

num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]

correlation(num_friends_good, daily_minutes_good) # 0.57
```

排除了这个异常值，相关性明显增强了（见图 5-3）。

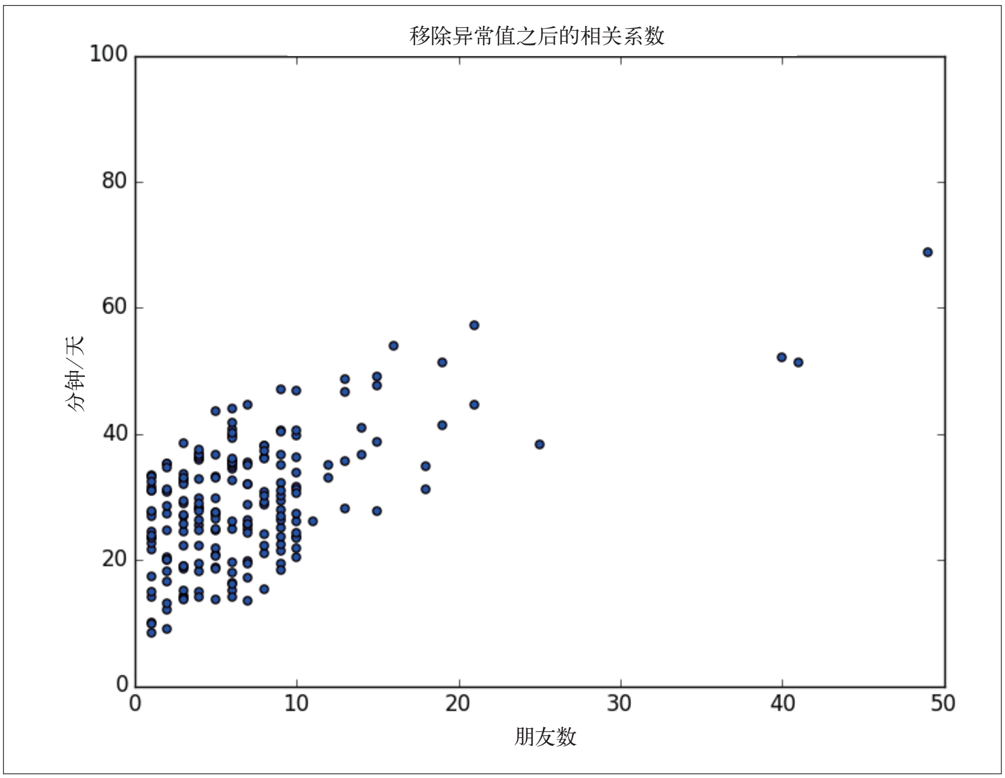


图 5-3：移除异常值之后的相关系数

通过进一步调查，你发现这个异常值实际上仅仅是一个内部测试账号，因而没人对移除它有异议。这样你就可以理直气壮地删除它了。

5.3 辛普森悖论

辛普森悖论是指分析数据时可能会发生的意外。具体而言，如果忽略了混杂变量，相关系

数会有误导性。

例如，假设你先将所有会员分成东海岸数据科学家和西海岸数据科学家两类，然后决定验证一下哪一边海岸的数据科学家更友好：

海岸	成员数	平均朋友数
西海岸	101	8.2
东海岸	103	6.5

很明显，西海岸的数据科学家比东海岸的数据科学家更招人喜欢。你的同事还可以给出许多理由解释这个结果：或许是阳光、咖啡、有机农产品，又或许是旖旎的太平洋风光。

但分析数据时，你却发现了一些奇怪的结论。如果你仅仅比较拥有博士学位的数据科学家，结论表明东海岸数据科学家的平均朋友数更多。如果再仅仅比较没有博士学位的数据科学家，结论仍然是东海岸的数据科学家平均拥有更多的朋友！

海岸	学位	成员数	平均朋友数
西海岸	博士	35	3.1
东海岸	博士	70	3.2
西海岸	非博士	66	10.9
东海岸	非博士	33	13.4

一旦你考虑了用户的学位，得出的相关系数就会发生变化！将东海岸科学家的数据和西海岸科学家的数据混同起来，会掩盖一件事实，即东海岸数据科学家更偏向博士类型。

这种现象在现实世界中时有发生。关键点在于，相关系数假设在*其他条件都相同*的前提之下衡量两个变量的关系。而当数据类型变成随机分配，就像置身于精心设计的实验之中时，“其他条件都相同”也许还不是一个糟糕的前提假设。但如果存在另一种类型分配的更深的机制，“其他条件都相同”可能会成为一个糟糕的前提假设。

避免这种窘境的唯一务实的做法是充分*了解你的数据*，并且尽可能核查所有可能的混杂因素。显然，这不可能万无一失。如果你没有这 200 个数据科学家的受教育程度的数据，你很可能就已经得出了西海岸的数据科学家天生更有社交能力的结论。

5.4 相关系数其他注意事项

相关系数为零表示两个变量之间不存在线性关系。但它们之间还可能会存在其他形式的关系。例如，如果：

```
x = [-2, -1, 0, 1, 2]
y = [ 2,  1, 0, 1, 2]
```

那么，x 和 y 的相关系数为 0。但容易看出，x 和 y 之间显然具有某种关系——y 中的每个元素等于 x 中相应元素的绝对值。然而，有一种关系它们却无法给出，即 x_i 和 mean(x) 之间的关系与 y_i 和 mean(y) 之间的关系并没有太大关联。这是一种相关系数试图捕捉的关系。

此外，相关系数无法告诉你关系有多强。例如：

```
x = [-2, 1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

以上这两个变量完全相关，但（这取决于你想度量什么）很有可能这种关系并没有实际意义。

5.5　相关和因果

你很可能听说过这样一句话："相关不是因果。"这样的说辞大致出自一位遇到了一堆威胁着他不可动摇的世界观的数据的人之口。然而，这是个重要的论断——如果 x 和 y 强相关，那么意味着可能 x 引起了 y，或 y 引起了 x，或者两者相互引起了对方，或者存在第三方因素同时引起了 x 和 y，或者什么都不是。

回想一下 num_friends 和 daily_minutes 之间的关系。如果 DataSciencester 用户在网站上拥有更多的朋友，可能会*引起*一个结果，即这些用户可能就会愿意在网上花费更多的时间。也可能是这种情形：如果每个朋友每天发布一定数量的内容，那么用户的朋友越多，就需要越多的时间来浏览朋友们的更新。

但是，也有这样一种可能。你泡在 DataSciencester 论坛上的时间越长，你就越有可能碰上和结识志同道合的朋友。这也意味着，在网站上花费时间越多，就会拥有更多朋友。

第三种可能是，越是那些热衷于数据科学的用户，就越喜欢在网上花更多时间（因为他们发现这更有趣），并且更乐于结交数据科学家朋友（因为他们对其他人不感冒）。

进行随机试验是证实因果关系的可靠性的一个好方法。你可以先将一组具有类似的统计数据的用户随机分为两组，再对其中一组施加稍微不同的影响因素，然后你会发现，不同的因素会导致不同的结果。

比如，如果你不介意因拿用户做试验而受谴责（http://www.nytimes.com/2014/06/30/technology/facebook-tinkers-with-users-emotions-in-news-feed-experiment-stirring-outcry.html?_r=0），可以随机从用户中抽取一个小样本，只给他们看他们一小部分朋友的动态更新。如果这个小样本中的用户在网上花费的时间相应地变少，那么你就可以肯定"拥有更多朋友会*引起*上网时间变长"这一结论了。

5.6 延伸学习

- SciPy（http://docs.scipy.org/doc/scipy/reference/stats.html）、pandas（http://pandas.pydata.org/）和 StatsModels（http://statsmodels.sourceforge.net/）等软件都含有丰富的统计函数。
- 统计学非常重要。（或者说，各种统计资料非常重要？）如果你想成为一名优秀的数据科学家，最好先熟读一本统计学教科书。网上有很多免费资源，其中我比较喜欢的有：
  - *OpenIntro Statistics*（https://www.openintro.org/stat/textbook.php）
  - *OpenStax Introductory Statistics*（http://openstaxcollege.org/textbooks/introductory-statistics）

第 6 章

概率

概率法则大致上是真实的，但某些时候会很荒谬。

——爱德华 · 吉本

如果对概率论和相关的数学原理理解得不够多，从事数据科学工作就极其困难。类似于本书第 5 章讲解统计学的方式，我们在本章粗略地讲解一下概率论，基本上不对细节做深入探讨。

为了达到我们的目的，你得把概率论视为对从事件空间中抽取的事件的不确定性进行量化的一种方式。我们暂且不探究术语的技术内涵，而是用掷骰子的例子来理解它们。空间是指所有可能的结果的集合。这些结果的任何一部分就是一个事件，比如，“骰子掷出的点数为 1”“骰子掷出的点数是偶数”等都是事件。

我们用 $P(E)$ 来标记“事件 E 的概率”。

我们接下来用概率理论构建模型，并用概率理论计算模型。概率理论无处不在。

只要你感兴趣，可以深入挖掘概率理论蕴含的哲学原理（边喝啤酒边研究就更好了）。但本书中我们不深入研究。

6.1　不独立和独立

泛泛地讲，如果 E 发生意味着 F 发生（或者 F 发生意味着 E 发生），我们就称事件 E 与事件 F 为不相互独立（dependent）。反之，E 与 F 就相互独立（independent）。

举个例子，如果两次掷起一枚均匀的硬币，那么我们无法根据第一次掷硬币的结果是否是正面朝上来判断第二次的结果是否正面朝上。第一次掷硬币的结果和第二次掷硬币的结果，这两个事件就是独立的。相反，如果第一次掷硬币的结果是正面朝上，那么我们能很明显能判断出两次掷硬币是否都是反面朝上。（如果第一次掷硬币的结果是正面朝上，那么很明显两次掷硬币的结果不可能都是反面朝上。）因此，第一次结果正面朝上和两次结果不可能都是反面朝上，这两个事件就是不独立的。

从数学角度讲，事件 E 和事件 F 独立意味着两个事件同时发生的概率等于它们分别发生的概率的乘积：

$$P(E，F)=P(E)P(F)$$

在上例中，“第一次掷硬币正面朝上”的概率为 1/2，“两次掷硬币都是背面朝上”的概率为 1/4，但“第一次掷硬币正面朝上并且两次掷硬币都是背面朝上”的概率为 0。

6.2　条件概率

如果事件 E 与事件 F 独立，那么定义式如下：

$$P(E，F)=P(E)P(F)$$

如果两者不一定独立（并且 F 的概率不为零），那么 E 关于 F 的条件概率式如下：

$$P(E|F)=P(E，F)/P(F)$$

条件概率可以理解为，已知 F 发生，E 会发生的概率。

更常用的公式是上式的变形：

$$P(E，F)=P(E|F)P(F)$$

如果 E 和 F 独立，则上式应该表示为：

$$P(E|F)=P(E)$$

这个数学公式意味着，F 是否发生并不会影响 E 是否发生的概率。

举一个常见的关于一个有两个孩子（性别未知）的家庭的有趣例子。

如果我们假设：

(1) 每个孩子是男孩和是女孩的概率相同
(2) 第二个孩子的性别概率与第一个孩子的性别概率独立

那么，事件“没有女孩”的概率是 1/4，事件“一个男孩，一个女孩”的概率为 1/2，事件“两个女孩”的概率为 1/4。

现在，我们的问题是，事件 B“两个孩子都是女孩”关于事件 G“大孩子是女孩”的条件

概率是多少？用条件概率的定义式进行计算如下：

$$P(B|G)=P(B，G)/P(G)=P(B)/P(G)=1/2$$

事件 B 与 G 的交集（“两个孩子都是女孩并且大孩子是女孩”）刚好是事件 B 本身。（一旦你知道两个孩子都是女孩，那大孩子必然是女孩。）

这个结果大致上符合你的直觉。

我们接着再问，事件“两个孩子都是女孩”关于事件“至少一个孩子是女孩”（L）的条件概率是多少？出乎意料的是，结果异于前问。

与前问相同的是，事件 B 和事件 L 的交集（“两个孩子都是女孩，并且至少一个孩子是女孩”）刚好是事件 B。这意味着：

$$P(B|L)=P(B，L)/P(L)=P(B)/P(L)=1/3$$

为什么会有这样的结果？如果你已知至少一个孩子是女孩，那么这个家庭有一个男孩和一个女孩的概率是有两个女孩的两倍。

我们可以通过“生成”许多家庭来验证这个结论：

```
def random_kid():
    return random.choice(["boy", "girl"])

both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)
for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == "girl":
        older_girl += 1
    if older == "girl" and younger == "girl":
        both_girls += 1
    if older == "girl" or younger == "girl":
        either_girl += 1

print "P(both | older):", both_girls / older_girl    # 0.514 ~ 1/2
print "P(both | either): ", both_girls / either_girl # 0.342 ~ 1/3
```

6.3 贝叶斯定理

贝叶斯定理是数据科学家的最佳朋友之一，它是条件概率的某种逆运算。假设我们需要计算事件 E 基于已发生的事件 F 的条件概率，但我们已知的条件仅仅是事件 F 基于已发生的事件 E 的条件概率。两次利用条件概率的定义，可以得到下式：

$$P(E|F) = P(E, F)/P(F) = P(F|E)P(E)/P(F)$$

事件 F 可以分割为两个互不重合的事件“F 和 E 同时发生”与“F 发生 E 不发生”。我们用符号 $\neg E$ 指代“非 E”（即“E 没有发生”），有下式：

$$P(F) = P(F, E) + P(F, \neg E)$$

因此：

$$P(E|F) = P(F|E)P(E)/[P(F|E)P(E)+P(F|\neg E)P(\neg E)$$

上式为贝叶斯定理常用的表达方式。

贝叶斯定理常常用来证明为什么数据科学家比医生更聪明。假设有这样一种病，10 000 个人中会有一个得这个病。还假设有种针对该病的测试，具有 99% 的可能性能给出正确判断（如果患病，测试显示“有病”，如果健康，则显示“无病”）。

阳性的测试结果意味着什么呢？我们用 T 表示“测试结果阳性”，用 D 表示“你患有该病”。那么，根据贝叶斯定理，如果测试结果为阳性，那么你患有该病的概率是：

$$P(D|T) = P(T|D)P(D)/[P(T|D)P(D)+P(T|\neg D)P(\neg D)$$

我们知道，$P(T|D)$，即一个人测试结果为阳性并且本人实际患病的概率为 0.99。$P(D)$，即一个人实际患病的概率是 1/10 000=0.0001 。$P(T|\neg D)$，即一个不患病的人检测结果呈阳性的概率是 0.01 。$P(\neg D)$，即一个人实际上不患该病的概率为 0.9999 。如果将以上数据代入贝叶斯定理，可得：

$$P(D|T) = 0.98\%$$

结果表示，测试结果为阳性的人实际患病的概率不到 1%。

> 移除异常值之后的相关系数我们实际上假设了人们接受测试的概率或多或少都是随机的。如果只有表现出特定症状的人才会去接受测试，我们应该将表达式重新表达为基于事件“测试结果正常，并且表现出症状”的条件概率，这样计算出的结果会增高很多。

对于数据科学家来说，这是小菜一碟，但大部分医生会猜测 $P(D|T)$ 的值接近 2。

一个更直观的计算方式是，首先假设总体包括 1 百万个人。你预期其中 100 个人患有该病，而这 100 个人中会有 99 个测试结果显示阳性。另一方面，你认为 999 900 个人不患有该病，其中 9999 个人测试结果呈阳性。这意味着在（99+9999）个测试结果呈阳性的人中，你认为仅有 99 个人实际上患有该病。

6.4 随机变量

*随机变量*指这样一种变量，其可能的取值有一种联合概率分布。定义一个简单的随机变量：掷起一枚硬币，如果正面朝上，随机变量等于 1，如果背面朝上，随机变量等于 0。可以再定义更复杂些的随机变量，如掷起一枚硬币 10 次，查看正面朝上的次数，或者从 `range(10)` 取出的一个值，每个数值被取到的可能性都相等。

联合分布对变量实现每种可能值都赋予了概率。通过掷硬币得到的随机变量等于 0 的概率为 0.5，等于 1 的概率也为 0.5。从 `range(10)` 中生成随机变量的分布为从 0 到 9 之间的每个数值赋予 0.1 的概率。

我们有时会讨论一个随机变量的*期望值*，表示这个随机变量可能值的概率加权值。掷硬币随机变量的期望值为 1/2(=0 ＊ 1/2 + 1 ＊ 1/2)，而 `range(10)` 随机变量的期望值为 4.5。

随机变量也可以基于某些*条件*事件产生，就像其他事件一样。回忆 6.2 节“条件概率”中提到的双生子例子：如果 X 是表示女孩个数的随机变量，那么 X 等于 0 的概率为 1/4 ，等于 1 的概率为 1/2 ，等于 2 的概率为 1/4。

在已知至少一个孩子是女孩后，接着再定义一个新的随机变量 Y，表示基于这个条件的女孩的个数。Y 等于 1 的概率为 2/3，等于 2 的概率为 1/3 。还定义另一个新的随机变量 Z，表示基于大孩子是女孩的条件之上的女孩的个数，Z 等于 1 的概率为 1/2 ，等于 2 的概率为 1/2。

大部分情况下，我们会隐式使用随机变量的概念，即在使用时，并没有对此加以特别关注。如果仔细思考，可以看出其中的端倪。

6.5 连续分布

掷硬币对应的是*离散分布*（discrete distribution）——对离散的结果赋予正概率。我们常常希望对连续结果的分布进行建模。（对于我们的研究目的来说，这些结果最好都是实数，但实际中并不总是这样的）例如，*均匀分布*（uniform distribution）函数对 0 到 1 之间的所有值都赋予相同的*权重*（weight）。

因为 0 和 1 之间有无数个数字，因而对每个点而言，赋予的权重几乎是零。因此，我们用带*概率密度函数*（probability density function，pdf）的连续分布来表示概率，一个变量位于某个区间的概率等于概率密度函数在这个区间上的积分。

> 如果积分运算不直观，有一种更简单的理解方式：一个分布的密度函数为 f，如果 h 很小，则变量的值落在 x 与 $x + h$ 之间的概率接近 $h*f(x)$。

均匀分布的密度函数如下：

```
def uniform_pdf(x):
    return 1 if x >= 0 and x < 1 else 0
```

如你预期的，一个服从均匀分布的随机变量落在 0.2 和 0.3 之间的概率为 1/10。在 Python 中，`random.random()` 是按均匀分布生成的伪随机数的函数。

我们还常常对*累积分布函数*（cumulative distribution function，cdf）感兴趣，这个函数给出了一个随机变量小于等于某一特定值的概率。生成均匀分布的累积分布函数不难（见图 6-1）：

```
def uniform_cdf(x):
    "returns the probability that a uniform random variable is <= x"
    if x < 0:   return 0    # 均匀分布的随机变量不会小于0
    elif x < 1: return x    # e.g. P(X <= 0.4) = 0.4
    else:       return 1    # 均匀分布的随机变量总是小于1
```

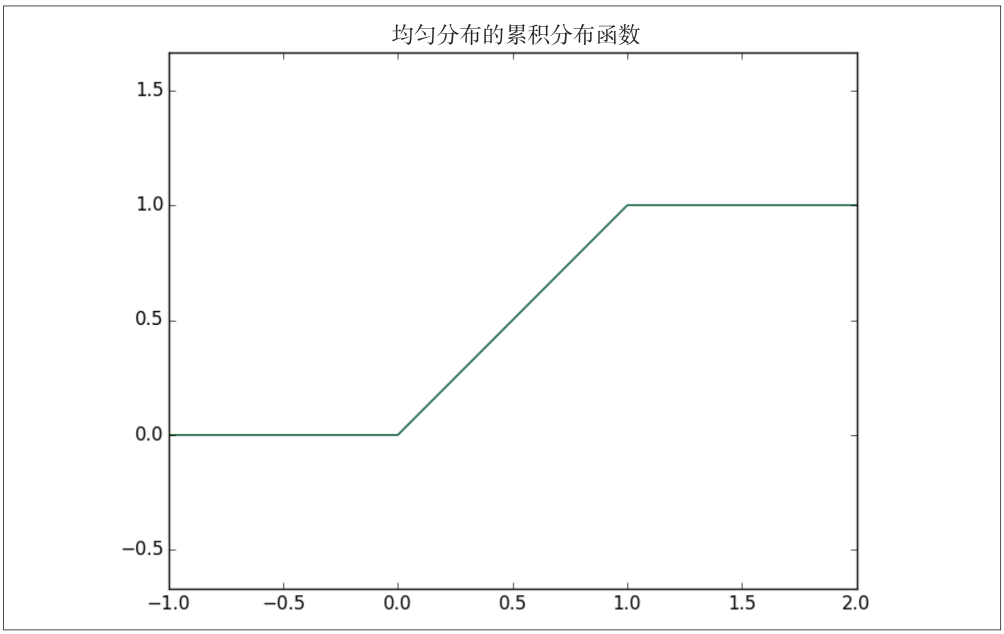


图 6-1：均匀分布的累积分布函数

6.6　正态分布

正态分布是分布之王！它是典型的钟型曲线形态分布函数，可以完全由两个参数决定：均值 μ（mu）和标准差 σ（sigma）。均值描述钟型曲线的中心，标准差描述曲线有多“宽”。

正态分布的分布函数如下：

$$f(x \mid \mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

我们可以这样实现：

```
def normal_pdf(x, mu=0, sigma=1):
    sqrt_two_pi = math.sqrt(2 * math.pi)
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (sqrt_two_pi * sigma))
```

在图 6-2 中我们绘出了这些概率密度函数，来看看它们的形状如何：

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_pdf(x,mu=-1)   for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend()
plt.title("多个正态分布的概率密度函数")
plt.show()
```

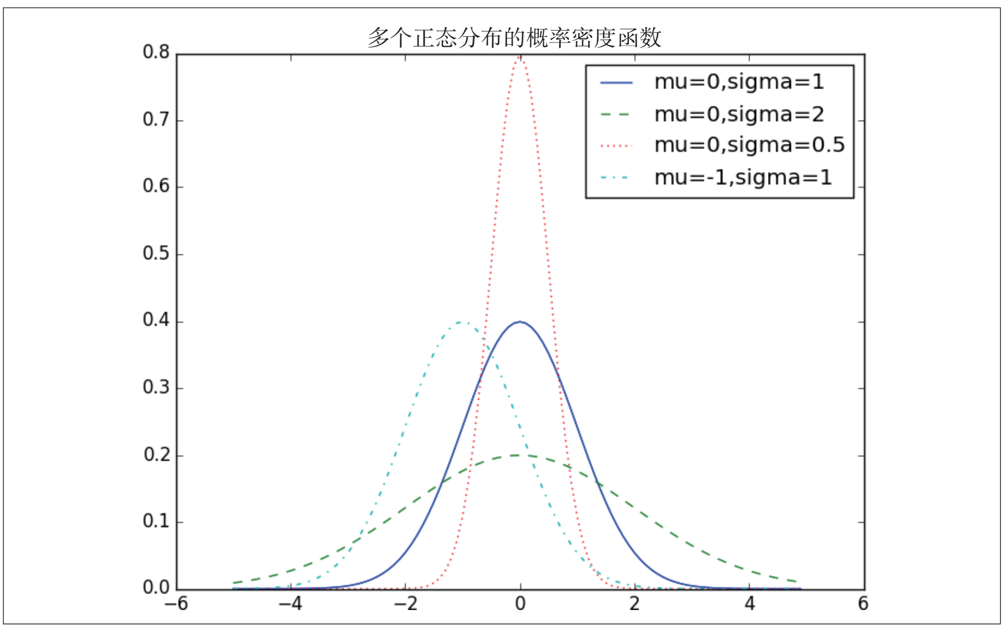


图 6-2：多个正态分布的概率密度函数

如果 μ=0 并且 σ=1，这个分布称为标准正态分布。如果 Z 是服从标准正态分布的随机变量，则有如下转换式：

$$X=\sigma Z+\mu$$

其中 X 也是正态分布，但均值是 μ，标准差是 σ。相反，如果 X 是均值为 μ 标准差为 σ 的正态分布，那么：

$$Z=(X-\mu)/\sigma$$

是标准正态分布的随机变量。

标准正态分布的累积分布函数无法用“基本”的解析形式表示，但在 Python 中可以用函数 `math.erf` 描述：

```
def normal_cdf(x, mu=0,sigma=1):
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

我们再绘出一系列概率累积分布函数（如图 6-3）：

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.5')
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # 底部右边
plt.title("多个正态分布的累积分布函数")
plt.show()
```

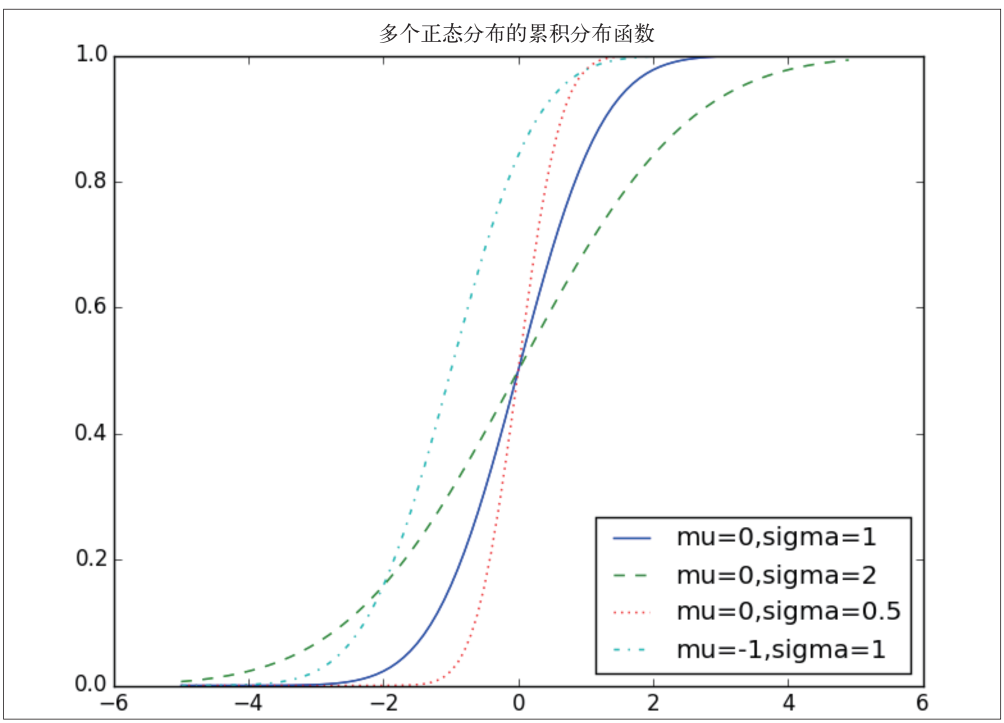


图 6-3：多个正态分布的累积分布函数

我们有时会需要对 normal_cdf 取逆，从而可以求出特定的概率的相应值。不存在计算逆函数的简便方法，但由于 normal_cdf 连续且严格递增，因而我们可以使用二分查找（https://en.wikipedia.org/wiki/Binary_search_algorithm）的方法：

```
def inverse_normal_cdf(p, mu=0, sigma=1, tolerance=0.00001):
    """find approximate inverse using binary search"""
    # 如果非标准型,先调整单位使之服从标准型
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)

    low_z, low_p = -10.0, 0            # normal_cdf(-10)是(非常接近)0
    hi_z,  hi_p  =  10.0, 1            # normal_cdf(10)是(非常接近)1
    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2     # 考虑中点
        mid_p = normal_cdf(mid_z)      # 和cdf在那里的值
        if mid_p < p:
            # midpoint仍然太低,搜索比它大的值
            low_z, low_p = mid_z, mid_p
        elif mid_p > p:
            # midpoint仍然太高,搜索比它小的值
            hi_z, hi_p = mid_z, mid_p
        else:
            break

    return mid_z
```

这个函数反复分割区间，直到分割到一个足够接近于期望概率的精细的 Z 值。

6.7 中心极限定理

正态分布的运用如此广泛，很大程度上归功于*中心极限定理*（central limit theorem）。这个定理说，一个定义为大量独立同分布的随机变量的均值的随机变量本身就是接近于正态分布的。

特别地，如果 $x_1,\cdots,x_n$ 都是均值为 μ、标准差为 σ 的随机变量，且 n 很大，那么：

$$\frac{1}{n}(x_1+\cdots+x_n)$$

近似正态分布，且均值为 μ，标准差为 $\sigma/\sqrt{n}$ 。等价于（其实更常用）：

$$\frac{(x_1+\cdots+x_n)-\mu n}{\sigma\sqrt{n}}$$

上式近似正态分布，均值为 0 ，标准差为 1。

举一个易于理解的验证例子——带有 n 和 p 两个参数的*二项式*随机变量。一个二项式随机变量 Binonimal(n,p) 是 n 个独立伯努利随机变量 Bernoulli(p) 之和，每个伯努利随机变量等

于 1 的概率是 p，等于 0 的概率是 $1-p$：

```
def bernoulli_trial(p):
    return 1 if random.random() < p else 0

def binomial(n, p):
    return sum(bernoulli_trial(p) for _ in range(n))
```

每个伯努利随机变量 Bernoulli(p) 的均值为 p，标准差为$\sqrt{p(1-p)}$。根据中心极限定理，当 n 变得很大，一个二项式随机变量 Binonimal(n,p) 近似于一个正态分布的随机变量，其中均值为 $\mu=np$，标准差为$\sigma=\sqrt{np(1-p)}$。如果把两个分布都在图上绘出来，很容易看出相似性：

```
def make_hist(p, n, num_points):

    data = [binomial(n, p) for _ in range(num_points)]

    # 用条形图绘出实际的二项式样本
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in histogram.values()],
            0.8,
            color='0.75')

    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))

    # 用线形图绘出正态近似
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]
    plt.plot(xs,ys)
    plt.title("二项分布与正态近似")
    plt.show()
```

比如，若调用函数 `make_hist(0.75, 100, 10000)`，可以得到图 6-4 中的图。

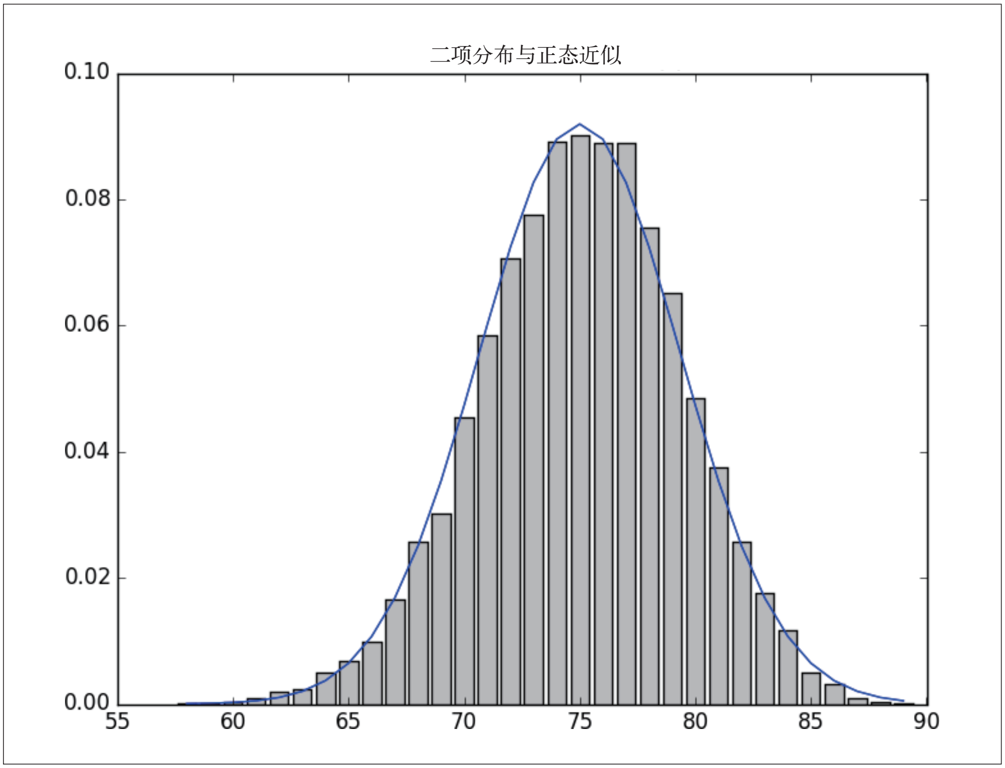


图 6-4：`make_hist` 的结果

近似表达的意义在于，如果你想知道掷起一枚均匀的硬币 100 次中正面朝上超过 60 次的概率，那么可以用一个正态分布 Normal(50, 5) 的随机变量大于 60 的概率来估计。这比计算二项式分布 Binonimal(100, 0.5) 的累积分布函数更容易（尽管在大多数应用中，你可以借助统计软件方便地计算出任何你想要的概率）。

6.8 延伸学习

- scipy.stats（http://docs.scipy.org/doc/scipy/reference/stats.html）包括绝大多数常见概率分布的概率密度函数和累积分布函数。
- 在第 5 章末我曾建议读者学习统计学教材，同样，这里我也建议读者学习概率论教材。我所知道的最好的在线教材是 *Introduction to Probability*（http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/amsbook.mac.pdf）。

第 7 章

假设与推断

深谙统计之道，方为人中之龙。

——萧伯纳

具备以上统计学和概率理论知识以后，我们接着该做什么呢？数据科学的科学部分，乃是不断针对我们的数据和生成数据的机制建立假设和检验假设。

7.1 统计假设检验

作为数据科学家，我们常常需要检验某个假设是否成立。有时，假设是诸如“这枚硬币是均匀的”“数据科学家喜欢 Python 胜过 R”或“如果人们点开某个突然弹出的小广告，广告的关闭按钮又小又难找，那么大家更倾向于离开这个页面，压根不会阅读”等可以被翻译成统计数据的断言。在各种各样的假设之下，这些统计数据可以理解为从某种已知分布中抽取的随机变量观测值，这可以让我们对这些假设是否成立做出论断。

典型的步骤是这样的，首先我们有一个零假设 H_0，它代表一个默认的立场，而替代假设 H_1 代表我们希望与零假设对比的立场。我们通过统计来决定我们是否可以拒绝 H_0，即判断它是否错误。通过举例能更直观地说明这个过程。

7.2 案例：掷硬币

假设有一枚硬币，我们试图判断它是否均匀，即任何一面朝上的可能性是否相等。首先，假设硬币落地后正面朝上的概率为 p，所以我们的零假设为硬币均匀，即 p=0.5。我们要对

比替代假设 $p \neq 0.5$ 来检验这个假设。

具体来说，首先掷硬币 n 次，将出现正面朝上的次数记为 X。每次掷硬币都是一次伯努利试验，意味着 X 是二项式随机变量 Binomial(n,p)，（正如第 6 章中所讲到的）可以用正态分布来拟合：

```
def normal_approximation_to_binomial(n, p):
    """finds mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

只要一个随机变量服从正态分布，我们就可以用 `normal_cdf` 来计算出一个实现数值位于（或不在）某个特定区间的概率：

```
# 正态cdf是一个变量在一个阈值以下的概率
normal_probability_below = normal_cdf

# 如果它不在阈值以下,就在阈值之上
def normal_probability_above(lo, mu=0, sigma=1):
    return 1 - normal_cdf(lo, mu, sigma)

# 如果它小于hi但不比lo小,那么它在区间之内
def normal_probability_between(lo, hi, mu=0, sigma=1):
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# 如果不在区间之内,那么就在区间之外
def normal_probability_outside(lo, hi, mu=0, sigma=1):
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

或者反过来，找出非尾区域，或者找出均值两边的（对称）区域，这个区域恰好对应特定比例的可能性。比如，如果我们需要找出以均值为中心、覆盖 60% 可能性的区间，那我们需要找到两个截点，使上尾和下尾各覆盖 20% 的可能性（给中间留出 60%）：

```
def normal_upper_bound(probability, mu=0, sigma=1):
    """returns the z for which P(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability, mu=0, sigma=1):
    """returns the z for which P(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability, mu=0, sigma=1):
    """returns the symmetric (about the mean) bounds
    that contain the specified probability"""
    tail_probability = (1 - probability) / 2

    # 上界应有在它之上的tail_probability
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)

    # 下界应有在它之下的tail_probability
```

```
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)

    return lower_bound, upper_bound
```

具体来讲，首先我们选择掷硬币 n=1000 次。如果关于均匀的原假设是正确的，那么 X 近似服从正态分布，均值为 50，标准差为 15.8：

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

我们需要对显著性（significance）下定义——我们有多大的可能性犯第 1 类错误（“容错”）。在这种情况下，我们拒绝了原假设 H_0，但实际上原假设是正确的。出于历史上的某些原因，可能性的大小通常设定为 5% 或者 1%。本书在此选择 5%。

考虑这样的检验——如果 X 落在以下区间以外，就拒绝原假设 H_0：

```
normal_two_sided_bounds(0.95, mu_0, sigma_0)  # (469, 531)
```

假设 p 实际上等于 0.5（即，此时 H_0 成立），那么我们有 5% 的可能观测到 X 落在区间之外，这正是我们想要的显著性。换句话说，如果 H_0 为真，那么 20 次检验中大约有 19 次会得出正确的结果。

我们常常对检验的势（power）有兴趣，它指的是不犯第 2 类错误的概率。第 2 类错误指原假设 H_0 是错的，但我们的检验结果没有拒绝原假设（即“纳伪”）。为了衡量统计的势，我们需要精确衡量 H_0 是错的意味着什么。（仅仅知道 p 不是 0.5 不足以为 X 的分布提供足够的信息。）具体来说，假如 p 实际上是 0.55，那么掷硬币的结果会稍微多偏向正面朝上。

在这种情形下，我们这样计算检验的势：

```
# 基于假设p是0.5时95%的边界
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# 基于p = 0.55的真实mu和sigma
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)

# 第2类错误意味着我们没有拒绝原假设
# 这会在X仍然在最初的区间时发生
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.887
```

如果我们把原假设变为掷硬币的结果不会偏重于正面朝上，即 $P \leqslant 0.5$，在这种情形下，我们使用单边检验。如果 X 远大于 50，我们就拒绝原假设，如果 X 小于 50，就不拒绝原假设。因此，显著性为 5% 的检验需要使用 `normal_probability_below` 来找出小于 95% 的概率对应的截点：

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# 是526 (< 531，因为我们在上尾需要更多的概率)
```

```
    type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
    power = 1 - type_2_probability      # 0.936
```

这是更有效的检验。如果 X 小于 469，我们就不再拒绝 H_0（如果 H_1 为真，这不太可能发生），当 X 在 526 和 531 之间时则拒绝 H_0（如果 H_1 为真，这很有可能发生）。

进行上述检验的另一种方式涉及 p 值。我们不再基于某个概率截点选择临界值，而是计算概率——假设 H_0 正确——我们可以找到一个至少与我们实际观测到的值一样极端的值。

对于硬币是否均匀的双面检验，我们可以做以下计算：

```
def two_sided_p_value(x, mu=0, sigma=1):
    if x >= mu:
        # 如果x大于均值,tail表示比x大多少
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # 如果x比均值小,tail表示比x小多少
        return 2 * normal_probability_below(x, mu, sigma)
```

如果我们希望看到结果中有 530 次为正面朝上，可以这样计算：

```
two_sided_p_value(529.5, mu_0, sigma_0)  # 0.062
```

为什么我们用 529.5 而不用 530？这就是所谓的连续校正（continuity correction）。它反映了一个事实，即对从掷硬币结果中看到 530 次正面朝上的概率而言，`normal_probability_between(529.5, 530.5, mu_0,sigma_0)` 是比 `normal_probability_between(530, 531, mu_0, sigma_0)` 更好的估计。

相应地，`normal_probability_above(529.5, mu_0, sigma_0)` 是看到至少 530 次正面朝上概率的更好估计。你可以在通过代码生成的图 6-4 中看到。

验证这种观点是否合理的一个方法是模拟：

```
extreme_value_count = 0
for _ in range(100000):
    num_heads = sum(1 if random.random() < 0.5 else 0    # 正面朝上的计数
                    for _ in range(1000))                # 在1000次抛掷中
    if num_heads >= 530 or num_heads <= 470:             # 并计算达到极值的频率
        extreme_value_count += 1                         # 极值的频率

print extreme_value_count / 100000   # 0.062
```

因为 p 值大于 5% 的显著性，所以我们不能拒绝原假设。如果我们看到了 532 次正面朝上，那么相应的 p 值为：

```
two_sided_p_value(531.5, mu_0, sigma_0)   # 0.0463
```

它小于 5% 的显著性，因此我们拒绝原假设。它正好是和之前相同的检验，只是计算统计

量的方法稍有不同。

同样，我们有：

```
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

对于单边检验，如果我们看到 525 次正面朝上，那么可以计算：

```
upper_p_value(524.5, mu_0, sigma_0)   # 0.061
```

这意味着我们会拒绝原假设。如果我们看到 527 次正面朝上，相应计算如下：

```
upper_p_value(526.5, mu_0, sigma_0)   # 0.047
```

根据结果，我们会拒绝原假设。

> 在调用函数 `normal_probability_above` 计算 p 值之前，需要确定你的数据大致上服从正态分布。数据科学的不良数据记录中充斥着差之毫厘失之千里的例子，原因在于“数据是正态分布的”，如果数据本身不是正态分布，那结果就毫无意义。
>
> 对正态分布的检验方法有好几种，绘图是不错的首选方案。

7.3 置信区间

我们一直在对正面朝上的概率 p 进行假设检验，这是未知的“正面朝上”分布的参数。对假设检验，我们还有第三种方法：在参数的观测值附近建立置信区间（confidence interval）。

例如，我们可以通过计算每次抛掷对应的伯努利随机变量的均值来估计不均匀硬币的概率——正面朝上记为 1，背面朝上记为 0，取这一系列伯努利随机变量的平均值。如果我们观测的 1000 次抛掷中有 525 次正面朝上，那么我们可以估计出 p 等于 0.525。

但是这个估计的可信度有多大呢？如果我们已知 p 的精确值，那么根据中心极限定理（见 6.7 节），伯努利随机变量的均值近似服从正态分布，其中均值为 p，标准为：

```
math.sqrt(p * (1 - p) / 1000)
```

这里，p 是未知的，所以我们使用估值：

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)   # 0.0158
```

这种计算是不严格的，但运用广泛。借助正态近似我们得出结论：以下区间包含真实参数 p 的可能性为 95%：

```
normal_two_sided_bounds(0.95, mu, sigma)        # [0.4940, 0.5560]
```

这是关于区间的解释，不是关于 p 值的解释。你需要这样理解：如果你重复实验很多次，其中 95% 的“真”参数（每次都相同）会落在观测到的置信区间（每次可能会都不同）中。

注意，我们没有得出不均匀硬币的结论，因为 0.5 落入了我们的置信区间。

如果我们观察到的是 540 次正面朝上，那么相应计算为：

```
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
normal_two_sided_bounds(0.95, mu, sigma) # [0.5091, 0.5709]
```

在这种情形下，“均匀硬币”没有落入置信区间。（“均匀硬币”的假设没有通过检验。如果假设是真的，那需要在 95% 的时间中都能通过。）

7.4 P-hacking

如果一个程序仅有 5% 的时间错误地拒绝了原假设，那么根据定义，5% 的时间会错误地拒绝原假设：

```
def run_experiment():
    """flip a fair coin 1000 times, True = heads, False = tails"""
    return [random.random() < 0.5 for _ in range(1000)]

def reject_fairness(experiment):
    """using the 5% significance levels"""
    num_heads = len([flip for flip in experiment if flip])
    return num_heads < 469 or num_heads > 531

random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
                      for experiment in experiments
                      if reject_fairness(experiment)])

print num_rejections   # 46
```

这意味着如果你有意找出“显著”结果，那么总是可以的。只要对数据的假设检验次数足够多，就总有某些会表现出显著性。移除右边的那些异常值，你就可以得到小于 0.05 的 p 值。（注意，这与 5.2 节讲的相关性有些类似。）

这就是所谓的 P-hacking（http://www.nature.com/news/scientific-method-statistical-errors-1.14700），它某种程度上是“基于 p 值框架的推断”的结果。批评这种方法的一篇绝好文章是“地球是圆的”（http://ist-socrates.berkeley.edu/~maccoun/PP279_Cohen1.pdf）。

如果想做好科学工作，就要在审查数据之前确定你的假设，就要在做假设之前整理好你的数据，并且要牢记，p 值并不是靠直觉得出的。（一个替代方案是下文 7.6 节“贝叶斯推断”。）

7.5 案例：运行A/B测试

你在 DataSciencester 的主要职责之一是经验值优化，这是个委婉的说法，其实就是设法让人点击广告。你的一个广告商针对数据科学家开发了一种新的能量饮料，广告部门的副总希望你帮助他在广告 A（“口味好”）和广告 B（“营养均衡”）之间进行选择。

作为一名科学家，你得做一次实验，对网站访问者随机放送不同的广告，并记录每个广告的点击数。

如果 1000 个看到广告 A 的人中有 990 个人点击广告，而 1000 个看到广告 B 的人中只有 10 个点击，你可以确认 A 是更棒的广告。但倘若区别并不如此分明，可以使用统计推断进行选择。

假设有 N_A 个人看到广告 A，其中 n_A 个人点击广告。每次广告浏览都是一次伯努利试验，其中 p_A 是点击广告 A 的概率。然后（如果 N_A 足够大。此处就足够大）我们知道 n_A/N_A 是近似服从正态分布的随机变量，其中均值为 p_A，标准差为$\sigma_A = \sqrt{p_A(1 - p_A)/N_A}$。同样，$n_B/N_B$ 是近似服从正态分布的随机变量，均值为 p_B，标准差为$\sigma_B = \sqrt{p_B(1 - p_B)/N_B}$：

```
def estimated_parameters(N, n):
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

如果我们假设这两个正态分布互相独立（这个假设是合理的，因为每个伯努利试验也是独立的），那么它们的差也是正态分布的，其中均值为 $p_B - p_A$，标准差为$\sqrt{\sigma_A^2 + \sigma_B^2}$。

> 这某种程度上有些欺骗性。只有在标准差已知的条件下数学推理才正确。我们从数据中估计参数，这意味着我们实际中应该用 t 分布。但如果数据集足够大，正态分布和 t 分布之间的差别可以忽略不计。

这意味着我们可以检验 p_A 和 p_B 相等（即 p_A-p_B 等于零）这个原假设，具体方式如下：

```python
def a_b_test_statistic(N_A, n_A, N_B, n_B):
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

这应该近似一个标准正态分布。

比如，如果“口味好”的广告从 1000 次浏览中获得 200 次点击量，而“营养均衡”广告则从 1000 次浏览中获得 180 次点击量，则统计量等于：

```python
z = a_b_test_statistic(1000, 200, 1000, 180)   # -1.14
```

如果两个均值实际上相等，那么看到如此大的差异的概率为：

```python
two_sided_p_value(z)                           # 0.254
```

这计算出的数值很大，以至于你不可以得出有差距的结论。另一方面，如果“营养均衡”仅仅获得 150 次点击量，则：

```python
z = a_b_test_statistic(1000, 200, 1000, 150)   # -2.94
two_sided_p_value(z)                           # 0.003
```

这意味着如果广告的效果相同，那么你看到有明显差异的概率只有 0.003。

7.6 贝叶斯推断

我们所看到的处理方式都包含对检验所做的与概率有关的陈述：“如果原假设正确，那么你观测到极端统计量的概率仅有 3%。”

推断的一个替代方法是将未知参数视为随机变量。分析师（也就是你）从参数的*先验分布*（prior distribution）出发，再利用观测数据和贝叶斯定理计算出更新后的*后验分布*（posterior distribution）。不再对检验本身给出概率判断，而是对参数本身给出概率判断。

比如，如果未知参数是概率（就像掷硬币的例子），我们使用 *Beta 分布*（Beta distribution）作为先验分布，Beta 分布仅对 0 和 1 赋值：

```python
def B(alpha, beta):
    """a normalizing constant so that the total probability is 1"""
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)

def beta_pdf(x, alpha, beta):
    if x < 0 or x > 1:          # [0, 1]之外没有权重
        return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```

一般来说，以上分布的权重中心为：

```python
alpha / (alpha + beta)
```

`alpha` 和 `beta` 越大，分布就越“紧密”。

例如，如果 `alpha` 和 `beta` 都是 1，那么刚好是均匀分布（以 0.5 为中心，非常分散）。如果 `alpha` 比 `beta` 大很多，那么大多数权重接近 1。如果 `alpha` 比 `beta` 小很多，那么大多数权重接近零。图 7-1 展示了几种不同的 Beta 分布。

让我们先假设一个先验分布 p。如果对硬币是否均匀不预设立场，那么将 `alpha` 与 `beta` 都设定为 1。或者如果我们坚信硬币有 55% 的可能正面朝上，就选择让 `alpha` 等于 55，`beta` 等于 45。

然后我们多次掷起硬币，结果有 h 次正面朝上，有 t 次背面朝上。根据贝叶斯定理（和一些太过冗繁的数学，此处不赘述），p 的先验分布仍然是 Beta 分布，但参数分别为 `alpha + h` 和 `beta + t`。

> 后验分布也是 Beta 分布，这并非偶然。二项分布给出了正面朝上的数字，Beta 是二项分布的共轭先验分布（conjugate prior，http://www.johndcook.com/blog/conjugate_prior_diagram/）。这意味着，无论你何时使用从相关的二项分布中得到的观测值更新 Beta 先验分布，你还是会得到一个 Beta 后验分布。

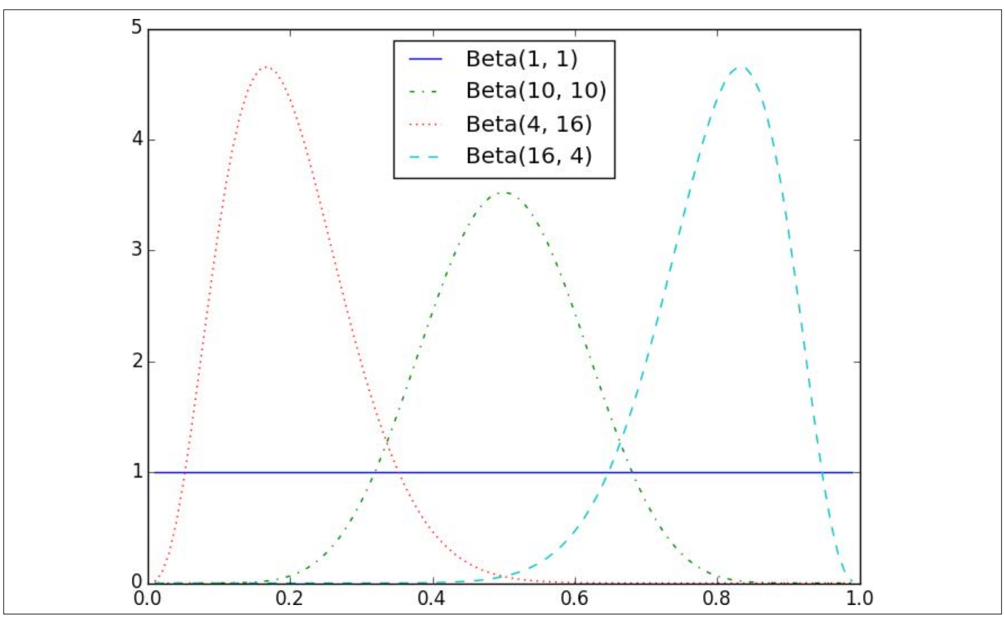


图 7-1：Beta 分布举例

假设你掷硬币 10 次并且观测到 3 次正面朝上。

如果你从均匀分布的先验开始（有时候不会采取硬币均匀的立场），那么你的后验分布为

Beta(4, 8)，中心为 0.33。如果你认为所有的可能性都相等，那么你最好的猜测就会非常接近观测到的概率。

如果你从 Beta(20, 20) 开始（表明硬币大致上是均匀的），那么你的后验分布为 Beta(23, 27)，中心为 0.46，这表明可能硬币稍稍倾向于背面朝上。

如果你从 Beta(30, 10) 开始（表明硬币是不均匀的，即有 75% 的可能会正面朝上），那么你的后验分布为 Beta(33, 17)，中心为 0.66。这种情况下，你仍然相信正面朝上的概率会大一些，只是没有一开始那么坚定了。这几个不同的后验分布如图 7-2 所示。

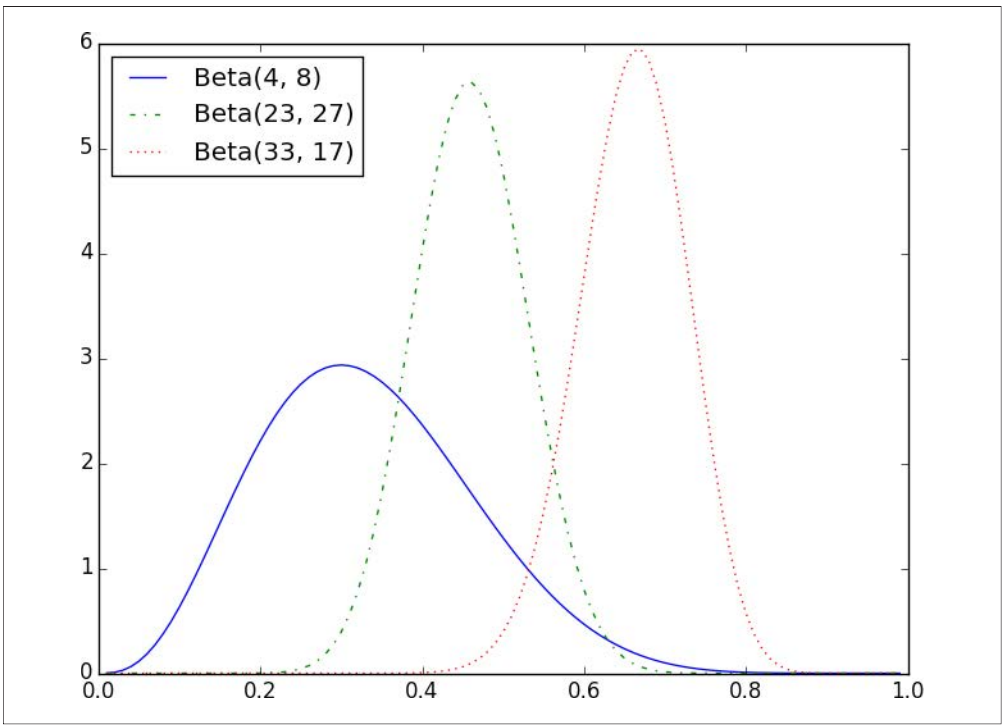


图 7-2：从不同先验分布得到的后验分布

如果你多次掷硬币，无论你最初选择了什么样的先验分布，先验分布对后验分布的影响会越来越小，直到最后得到（几乎）相同的后验分布。

比如，无论你最初对掷硬币的结果有怎样的倾向猜想，当看到 2000 次掷硬币的结果中有 1000 次正面朝上时，你都会很难维持原先的看法（除非你极端地选择了 Beta(1000000, 1) 这样的先验分布）。

有趣的是，这允许我们对假设“基于先验分布和已观测数据，正面朝上的概率介于 49% ~ 51% 的可能性仅有 5%”做出概率判断。这在哲学上不同于论断“如果硬币是均匀的，那

么只有 5% 的机会能观测到极端数据”。

用贝叶斯推断进行假设检验是饱受争议的——部分源于它的数学原理非常复杂，部分源于选择先验分布的主观性。本书中我们不会挖掘得太深，但稍作了解还是有益的。

7.7 延伸学习

- 我们仅仅讲解了统计推断的一点皮毛知识。第 5 章末推荐的相关书籍有大量更加深入的细节知识。
- 在线公开课提供了涵盖许多相关主题的数据分析与统计推断课程（https://www.coursera.org/course/statistics）。

第 8 章

梯度下降

夸耀自己血统者，实际上夸耀的是他们对别人的亏欠。

——塞内加

从事数据科学工作时，常常会面临这样的需要：为某种特定的情形寻找最佳模型。“最佳”常常会被解读为某种类似于“最小化模型残差”或者“最大化数据的可能性”。换句话说，它代表了优化某种问题的解决方案。

这意味着我们需要解决一连串的最优化问题。尤其是，我们需要从零开始解决问题。我们采用的方法是一种叫作*梯度下降*（gradient descent）的技术，适合从零开始逐步解决问题。也许你无法从中体味出兴奋感，但它会在本书中教我们做很多令人兴奋的事情，所以，务必跟随我。

8.1　梯度下降的思想

假设我们拥有某个函数 f，这个函数输入一个实数向量，输出一个实数。一个简单的例子如下：

```
def sum_of_squares(v):
    """computes the sum of squared elements in v"""
    return sum(v_i ** 2 for v_i in v)
```

我们常常需要最大化（或最小化）这个函数。这意味着我们需要找出能计算出最大（或最小）可能值的输入 v。

对我们的函数来说，梯度（在微积分里，这表示偏导数向量）给出了输入值的方向，在这个方向上，函数增长得最快。（如果记不起微积分，用我提到的关键词上网查查。）

相应地，最大化函数的算法首先从一个随机初始点开始，计算梯度，在梯度方向（这是使函数增长最快的一个方向）上跨越一小步，再从一个新的初始点开始重复这个过程。同样，你也可以在相反方向上逐步最小化函数，如图 8-1 所示。

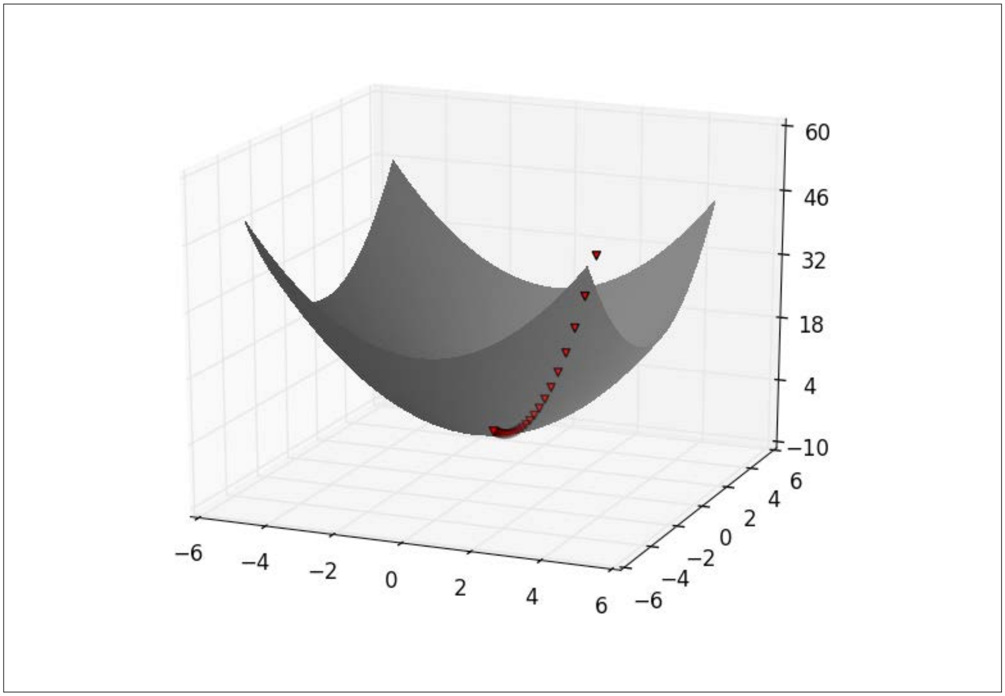
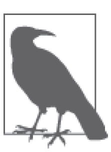


图 8-1：用梯度下降法计算最小点



如果一个函数有一个全局最小点，那么这个方法很可能会找到它。如果这个函数有多个（局部）最小点，那么这种方法可能找不到这个点，但你可以通过多尝试一些初始点来重复运行这个方法。如果一个函数没有最小点，也许计算会陷入死循环。

8.2 估算梯度

如果 `f` 是单变量函数，那么它在 `x` 点的导数衡量了当 `x` 发生变化时，`f(x)` 变化了多少。导数通过差商的极限来定义：

```
def difference_quotient(f, x, h):
    return (f(x + h) - f(x)) / h
```

其中 h 趋近于 0。

（许多微积分初学者常常受困于极限的数学定义。这里我们不妨说，你认为它是什么，它就是什么。）

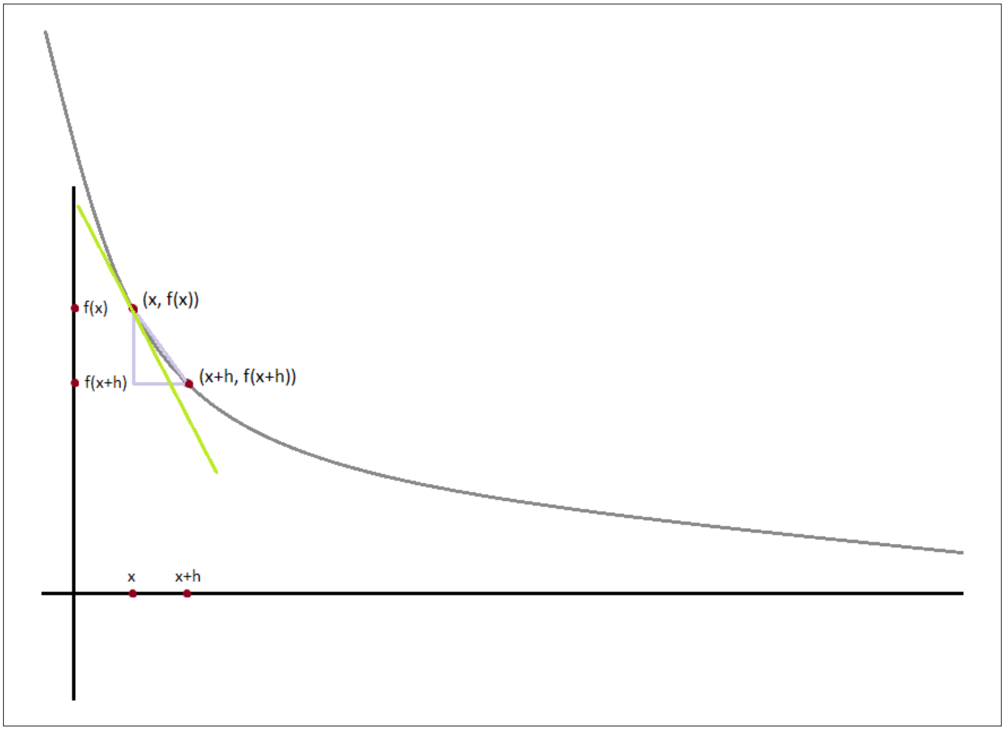


图 8-2：通过差商来求近似导数

导数就是在点 $(x, f(x))$ 的切线的斜率，而差商就是通过点 $(x, f(x))$ 和点 $(x+h, f(x+h))$ 的割线的斜率。当 h 越来越小，割线与切线就越来越接近（见图 8-2）。

很多函数可以精确地计算导数，比如平方函数 square：

```
def square(x):
    return x * x
```

它的导数为：

```
def derivative(x):
    return 2 * x
```

你可以通过计算来确认——如果你想的话——先显式地计算差商，再取极限。

如果算不出梯度（或不想算）呢？ Python 中无法直接运算极限，但可以通过计算一个很小

的变动 e 的差商来估算微分。图 8-3 给出了这个估算的结果：

```
derivative_estimate = partial(difference_quotient, square, h=0.00001)

# 绘出导入matplotlib.pyplot作为plt的基本相同的形态
x = range(-10,10)
plt.title("精确的导数值与估计值")
plt.plot(x, map(derivative, x), 'rx', label='Actual')            # 用 x 表示
plt.plot(x, map(derivative_estimate, x), 'b+', label='Estimate') # 用 + 表示
plt.legend(loc=9)
plt.show()
```

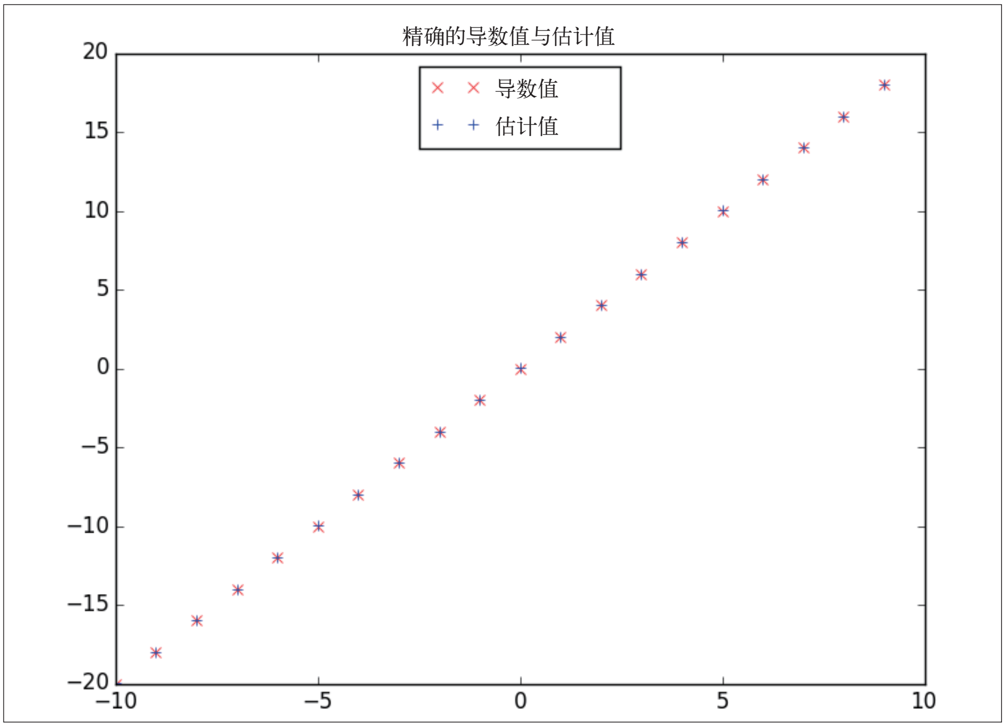


图 8-3：差商近似值的拟合度

当 f 是一个多变量函数时，它有多个偏导数，每个偏导数表示仅有一个输入变量发生微小变化时函数 f 的变化。

我们把导数看成是其第 *i* 个变量的函数，其他变量保持不变，以此来计算它第 *i* 个偏导数：

```
def partial_difference_quotient(f, v, i, h):
    """compute the ith partial difference quotient of f at v"""
    w = [v_j + (h if j == i else 0)    # 只对v的第i个元素增加h
         for j, v_j in enumerate(v)]

    return (f(w) - f(v)) / h
```

再以同样的方法估算它的梯度函数：

```
def estimate_gradient(f, v, h=0.00001):
    return [partial_difference_quotient(f, v, i, h)
            for i, _ in enumerate(v)]
```

> “差商估算法”的主要缺点是计算代价很大。如果 v 长度为 n，那么 estimate_gradient 为了计算 f 需要 $2n$ 个不同的输入变量。如果你需要反复计算梯度，那需要做很多额外的工作。

8.3 使用梯度

很容易看出，当输入 v 是零向量时，函数 sum_of_squares 取值最小。但如果不知道输入是什么，可以用梯度方法从所有的三维向量中找到最小值。我们先找出随机初始点，并在梯度的反方向以小步逐步前进，直到梯度变得非常非常小：

```
def step(v, direction, step_size):
    """move step_size in the direction from v"""
    return [v_i + step_size * direction_i
            for v_i, direction_i in zip(v, direction)]

def sum_of_squares_gradient(v):
    return [2 * v_i for v_i in v]

# 选取一个随机初始值
v = [random.randint(-10,10) for i in range(3)]

tolerance = 0.0000001

while True:
    gradient = sum_of_squares_gradient(v)   # 计算v的梯度
    next_v = step(v, gradient, -0.01)       # 取负的梯度步长
    if distance(next_v, v) < tolerance:     # 如果收敛了就停止
        break
    v = next_v                              # 如果没汇合就继续
```

如果运行以上程序，你会发现，它总是止于一个非常接近 [0,0,0] 的 v 值。tolerance 值设定得越小，v 值就越接近 [0,0,0]。

8.4 选择正确步长

尽管向梯度的反向移动的逻辑已经清楚了，但移动多少还不明了。事实上，选择合适的步长更像艺术而非科学。主流的选择方法有：

- 使用固定步长
- 随时间增长逐步减小步长

- 在每一步中通过最小化目标函数的值来选择合适的步长

最后一种方法看上去不错，但它的计算代价也最大。我们可以尝试一系列步长，并选出使目标函数值最小的那个步长来求其近似值：

```
step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
```

某些步长可能会导致函数的输入无效。所以，我们需要创建一个对无效输入值返回无限值（即这个值永远不会成为任何函数的最小值）的“安全应用”函数：

```
def safe(f):
    """return a new function that's the same as f,
    except that it outputs infinity whenever f produces an error"""
    def safe_f(*args, **kwargs):
        try:
            return f(*args, **kwargs)
        except:
            return float('inf')         # 意思是Python中的“无限值”
    return safe_f
```

8.5　综合

通常而言，我们有一些 target_fn 函数，需要对其进行最小化，也有梯度函数 gradient_fn。比如，函数 target_fn 可能代表模型的残差，它是参数的函数。我们可能需要找到能使残差尽可能小的参数。

此外，假设我们（以某种方式）为参数 theta_0 设定了某个初始值，那么可以如下使用梯度下降法：

```
def minimize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    """use gradient descent to find theta that minimizes target function"""

    step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]

    theta = theta_0                           # 设定theta为初始值
    target_fn = safe(target_fn)               # target_fn的安全版
    value = target_fn(theta)                  # 我们试图最小化的值

    while True:
        gradient = gradient_fn(theta)
        next_thetas = [step(theta, gradient, -step_size)
                       for step_size in step_sizes]

    # 选择一个使残差函数最小的值
    next_theta = min(next_thetas, key=target_fn)
    next_value = target_fn(next_theta)

    # 当“收敛”时停止
    if abs(value - next_value) < tolerance:
```

```
        return theta
    else:
        theta, value = next_theta, next_value
```

我们称它为 minimize_batch，因为在每一步梯度计算中，它都会搜索整个数据集（因为 target_fn 代表整个数据集的残差）。在下一部分中，我们会探讨另一种方法，一次仅考虑一个数据点。

有时候，我们需要最大化某个函数，这只需要最小化这个函数的负值（相应的梯度函数也需取负）：

```
def negate(f):
    """return a function that for any input x returns -f(x)"""
    return lambda *args, **kwargs: -f(*args, **kwargs)

def negate_all(f):
    """the same when f returns a list of numbers"""
    return lambda *args, **kwargs: [-y for y in f(*args, **kwargs)]

def maximize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    return minimize_batch(negate(target_fn),
                          negate_all(gradient_fn),
                          theta_0,
                          tolerance)
```

8.6 随机梯度下降法

正如之前提到的，我们常常用梯度下降的方法，通过最小化某种形式的残差来选择模型参数。如果使用之前的批处理方法，每个梯度计算步都需要我们预测并计算整个数据集的梯度，这使每一步都会耗费很长时间。

现在，这些残差函数常常具有*可加性*（additive），意味着整个数据集上的预测残差恰好是每个数据点的预测残差之和。

在这种情形下，我们转而使用一种称为*随机梯度下降*（stochastic gradient descent）的技术，它每次仅计算一个点的梯度（并向前跨一步）。这个计算会反复循环，直到达到一个停止点。

在每个循环中，我们都会在整个数据集上按照一个随机序列迭代：

```
def in_random_order(data):
    """generator that returns the elements of data in random order"""
    indexes = [i for i, _ in enumerate(data)]  # 生成索引列表
    random.shuffle(indexes)                    # 随机打乱数据
    for i in indexes:                          # 返回序列中的数据
        yield data[i]
```

我们对每个数据点都会进行一步梯度计算。这种方法留有这样一种可能性，即也许会在最小值附近一直循环下去，所以，每当停止获得改进，我们都会减小步长并最终退出：

```
def minimize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):

    data = zip(x, y)
    theta = theta_0                                  # 初始值猜测
    alpha = alpha_0                                  # 初始步长
    min_theta, min_value = None, float("inf")        # 迄今为止的最小值
    iterations_with_no_improvement = 0

    # 如果循环超过100次仍无改进,停止
    while iterations_with_no_improvement < 100:
        value = sum( target_fn(x_i, y_i, theta) for x_i, y_i in data )

        if value < min_value:
            # 如果找到新的最小值,记住它
            # 并返回到最初的步长
            min_theta, min_value = theta, value
            iterations_with_no_improvement = 0
            alpha = alpha_0
        else:
            # 尝试缩小步长,否则没有改进
            iterations_with_no_improvement += 1
            alpha *= 0.9

        # 在每个数据点上向梯度方向前进一步
        for x_i, y_i in in_random_order(data):
            gradient_i = gradient_fn(x_i, y_i, theta)
            theta = vector_subtract(theta, scalar_multiply(alpha, gradient_i))

    return min_theta
```

随机化通常比批处理化快很多。当然，我们也希望获得最大化的结果：

```
def maximize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):
    return minimize_stochastic(negate(target_fn),
                               negate_all(gradient_fn),
                               x, y, theta_0, alpha_0)
```

8.7 延伸学习

- 继续往下读，我们将用梯度下降法解决本书剩余部分提到的很多问题。
- 如果继续推荐你阅读教材，你该感到厌倦了。幸好，这次推荐你读的是 *Active Calculus*（http://scholarworks.gvsu.edu/books/10/），它似乎比我读过的任何微积分教材都要友好一些。
- scikit-learn 有一个随机梯度下降模块（http://scikit-learn.org/stable/modules/sgd.html），它在某些方面讲得比较笼统，而在其他一些方面讲得很详细。事实上，在真实世界的大多数场景中，你都会用到库，库中的优化技术已经充分考虑到背后的原理了，所以你不必为此操心（绝不会某个时候就不能正常工作了，毫无疑问，这不可能）。

第 9 章

获取数据

写作本书我用了三个月的时间；构思只用了三分钟；而收集书中的数据，则用了我的一生。

——F. 斯科特 · 菲兹杰拉德

为了成为一名数据科学家，你需要数据。事实上，作为数据科学家，你会花超大一部分时间来获取、清理和转换数据。必要时，你总可以自己输入数据（或者可以让你的助手来做），但通常这样做比较浪费时间。本章我们来看看利用 Python 获取数据并得到正确格式的不同方法。

9.1　stdin和stdout

如果在命令行运行 Python 脚本，你可以用 `sys.stdin` 和 `sys.stdout` 以管道（pipe）方式传递数据。例如，以下脚本按行读入文本，然后划分出和一个正则表达式匹配的行：

```
# egrep.py
import sys, re

# sys.argv是命令行参数的列表
# sys.argv[0]是程序自己的名字
# sys.argv[1]会是在命令行上指定的正则表达式
regex = sys.argv[1]

# 对传递到这个脚本中的每一个行
for line in sys.stdin:
    # 如果它匹配正则表达式,则把它写入stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

然后对收到的行计数并输出计数结果：

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

# 输出去向 sys.stdout
print count
```

你可以用这种方法来计数文件中有多少行包含数字。在 Windows 中，你可以用：

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

而在 Unix 系统中，你可以用：

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

“|”运算符是个管道字符，它的意思是“使用左边命令的输出作为右边命令的输入”。可以使用这种方法精心设计数据处理的管道。

如果你使用的是 Windows，你可以在这行命令中去掉所包含的 python 部分：

```
type SomeFile.txt | egrep.py "[0-9]" | line_count.py
```

如果你用 Unix 系统，这么做可能会需要多做一些额外工作。

类似地，下面的这个脚本计算了单词的数量并给出了最常用的单词：

```
# most_common_words.py
import sys
from collections import Counter

# 传递单词的个数作为第一个参数
try:
    num_words = int(sys.argv[1])
except:
    print "usage: most_common_words.py num_words"
    sys.exit(1)   # 非零的exit代码表明有错误

counter = Counter(word.lower()                      # 小写的单词
                  for line in sys.stdin
                  for word in line.strip().split() # 用空格划分
                  if word)                          # 跳过空的 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
```

```
        sys.stdout.write(word)
        sys.stdout.write("\n")
```

之后你可以对圣经文本使用这个脚本：

```
C:\DataScience>type the_bible.txt | python most_common_words.py 10
64193   the
51380   and
34753   of
13643   to
12799   that
12560   in
10263   he
9840    shall
8987    unto
8836    for
```

如果你是个 Unix 编程老手，你可能会很熟悉系统里许多内置的命令行工具（比如 egrep），但你自己从零开始创建这些工具或许更好一些。毕竟，用到的时候刚好知道，总是好的。

9.2　读取文件

可以显式地用代码来读写文件。用 Python 处理文件非常简便。

9.2.1　文本文件基础

处理文本文件的第一步是通过 open 命令来获取一个文件对象：

```
# 'r' 意味着只读
file_for_reading = open('reading_file.txt', 'r')

# 'w' 是写入——会破坏已存在的文件！
file_for_writing = open('writing_file.txt', 'w')

# 'a' 是添加——加入到文件的末尾
file_for_appending = open('appending_file.txt', 'a')

# 完成以后别忘了关闭文件
file_for_writing.close()
```

因为非常容易忘记关闭文件，所以你应该在 with 程序块里操作文件，这样在结尾处文件会被自动关闭：

```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)

# 此时,f已经关闭了,别再试图使用它
```

```
process(data)
```

如果需要读取一个完整的文本文件，可以使用 for 语句对文件的行进行迭代：

```
starts_with_hash = 0

with open('input.txt','r') as f:
    for line in file:              # 查找文件中的每一行
        if re.match("^#",line):    # 用正则表达式判断它是否以'#'开头
            starts_with_hash += 1  # 如果是,计数加1
```

按这种方法得到的每一行会用换行符来结尾，所以在对读入的行操作之前会经常需要用 strip() 来进行处理。

例如，假设你有一个写满电子邮件地址的文件，每个地址一行，你想利用这个文件生成域名的直方图。正确地提取域名的规则有些微妙（如公共后缀列表，https://publicsuffix.org/），但一个好的近似方案是只取出电子邮件地址中 @ 后面的部分。（对于像 joel@mail.datasciencester.com 这样的邮件地址，会给出错误的答案。）

```
def get_domain(email_address):
    """split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

9.2.2 限制的文件

我们刚刚处理的假想电子邮件地址文件每行只有一个地址。更常见的情况是你会处理每一行包含许多数据的文件。这种文件通常是用逗号分割或 tab 分割的，每一行有许多字段，用逗号（或 tab）来表示一个字段的结束和另一个字段的开始。

这开始变得复杂了，各字段中带有逗号、tab 和换行符（这是你不可避免地要处理的）。因为这个原因，几乎总是会犯的一个错误是你自己尝试去解析它们。相反，你应该使用 Python 的 csv 模块（或者 pandas 库）。出于微软方面（你可以对其大加责备）的技术原因，你应该总是通过把 *b* 包括在 *r* 或 *w* 之后来用二进制模式处理 csv 文件（见 Stack Overflow，http://stackoverflow.com/questions/4249185/using-python-to-append-csv-files）。

如果文件没有头部（意味着你可能想把每一行作为一个列表，这带来的麻烦是你需要知道每一列是什么），你可以使用 csv.reader 对行进行迭代，每一行都会被处理成恰当划分的列表。

例如，如果有这样一个用 tab 划分的股票价格文件：

```
6/20/2014   AAPL    90.91
6/20/2014   MSFT    41.68
6/20/2014   FB  64.5
6/19/2014   AAPL    91.86
6/19/2014   MSFT    41.51
6/19/2014   FB  64.34
```

我们可以用下面的程序块来处理：

```
import csv

with open('tab_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

如果文件存在头部：

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

你既可以跳过头部的行（利用对 `read.next()` 的初始调用）也可以利用 `csv.DictReader` 把每一行读成字典（把头部作为关键字）：

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.DictReader(f, delimiter=':')
    for row in reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(row["closing_price"])
        process(date, symbol, closing_price)
```

即使你的文件缺少头部，你仍可以通过把关键字作为文件名参数传输来使用 `DictReader`。

同样，你可以用 `csv.writer` 来写限制的文件：

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

如果行中的各字段本身包含逗号，`csv.writer` 可以正确处理。但你自己手动写成的则很可能不会正确处理。比如，如果你尝试这样做：

```
results = [["test1", "success", "Monday"],
```

```
            ["test2", "success, kind of", "Tuesday"],
            ["test3", "failure, kind of", "Wednesday"],
            ["test4", "failure, utter", "Thursday"]]

# 不要这么做!
with open('bad_csv.txt', 'wb') as f:
    for row in results:
        f.write(",".join(map(str, row))) # 可能包含了太多逗号!
        f.write("\n")                    # 行也可能会换行!
```

你最终会得到像下面这样一个 csv 文件：

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

没人能看懂它的意思。

9.3 网络抓取

另一种获取数据的方法是从网页抓取数据。获取一个网页十分容易，但从网页上抓取有意义的结构化信息就不那么容易了。

9.3.1 HTML和解析方法

网络上的页面是由 HTML 写成的，其中文本被（理想化地）标记为元素和它们的属性：

```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

在理想的情况下，所有的网页为我们方便地按语义标记，我们可以使用类似这样的规则来提取数据：找到 id 是 subject 的 <p> 元素并返回它所包含的文本。但在真实的世界中，HTML 并不总是具有很好的格式的，更不用说注解了。这意味着如果我们想搞清其含义，需要一些帮助。

为了从 HTML 里得到数据，我们需要使用 BeatifulSoup 库（http://www.crummy.com/software/BeautifulSoup/），它对来自网页的多种元素建立了树结构，并提供了简单的接口来获取它们。本书写作时，最新的版本是 Beatiful Soup 4.3.2（pip install beautifulsoup4），我们即将用到的就是这个版本。我们也会用到 requests 库（pip install requests，http://

docs.python-requests.org/en/latest/），它与内置在 Python 中的其他方法相比，是一种发起 HTTP 请求的更好的方式。

Python 内置的 HTML 解析器是有点严格的，这意味着它并不总是能处理那些没有很好地格式化的 HTML。因此，我们需要使用另外一种解析器，它需要先安装：

```
pip install html5lib
```

为了使用 Beatiful Soup，我们要把一些 HTML 传递给 BeautifulSoup() 函数。在我们的例子中，这些 HTML 是对 requests.get 进行调用的结果：

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

完成这个步骤之后，我们可以用一些简单的方法得到完美的解析。

通常我们会处理一些 Tag 对象，它们对应于 HTML 页面结构的标签表示。

比如，找到你能用的第一个 <p> 标签（及其内容）：

```
first_paragraph = soup.find('p')        # 或仅仅soup.p
```

可以对 Tag 使用它的 text 属性来得到文本内容：

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

另外可以把标签当作字典来提取其属性：

```
first_paragraph_id = soup.p['id']        # 如果没有'id'则报出KeyError
first_paragraph_id2 = soup.p.get('id')   # 如果没有'id'则返回None
```

可以一次得到多个标签：

```
all_paragraphs = soup.find_all('p')     # 或仅仅soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

通常你会想通过一个类（class）来找到标签：

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

此外，可以把这些方法组合起来运用更复杂的逻辑。比如，如果想找出包含在一个 <div> 元素中的每一个 <span> 元素，可以这么做：

```
# 警告,将多次返回同一个span元素
# 如果它位于多个div元素里
# 如果是这种情况,要更谨慎一些
spans_inside_divs = [span
                     for div in soup('div')     # 对页面上的每个<div>
                     for span in div('span')]   # 找到其中的每一个<span>
```

仅仅上述几个特性就可以帮助我们做很多事。如果你需要做更复杂的事情（或仅仅是出于好奇），那就去查看文档吧。

当然，无论多重要的数据，通常也不会标记成 `class="important"`。你需要仔细检查源 HTML，通过你选择的逻辑进行推理，并多考虑边界情况来确保数据的正确性。接下来我们看一个例子。

9.3.2 案例：关于数据的O'Reilly图书

DataSciencester 的某位潜在投资者认为数据只会风靡一时。为了证明他是错的，你打算查看一下 O'Reilly 出版社这些年来总共出版过多少数据类的图书。通过对 O'Reilly 网站的挖掘，你发现它有许多有关数据图书（以及视频）的页面，每次 30 个条目的目录页面有这样的 URL：

```
http://shop.oreilly.com/category/browse-subjects/data.do?
sortby=publicationDate&page=1
```

我们都不笨（而且也不想让自己的抓取器被封），所以在每次从网站抓取数据之前都该看一下这家网站是否有某种获取政策。 查看以下页面：

```
http://oreilly.com/terms/
```

看起来对这个项目没有明文禁止。但为了做一个守法的好公民，我们还应该查看一下 robots.txt 文件，看看一个网络抓取者要有怎样的行为规范。http://shop.oreilly.com/robots.txt 有以下重要内容：

```
Crawl-delay: 30
Request-rate: 1/30
```

第一行告诉我们应该在两次请求之间等待 30 秒，第二行告诉我们每 30 秒只能请求一个页面。所以从根本上说这两行原则讲的是同一件事。（文件里还有一些内容说明有些目录页是不能抓取的，但是我们的 URL 不在其中，所以我们可以放心了。）

O'Reilly 是有可能改变某些网站政策的，那样会打破本小节的所有逻辑。我会尽我所能预防这种情况的发生，当然，我对 O'Reilly 并没有太大的影响力。然而，如果你们每人都发动所有认识的人买一本这书的话……

为了弄清该怎样提取数据，让我们下载其中一个页面，把它传给 Beatiful Soup：

```
# 除非是写进书里,否则你没必要这样拆分一个url
url = "http://shop.oreilly.com/category/browse-subjects/" + \
      "data.do?sortby=publicationDate&page=1"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
```

如果你查看页面的源代码（在浏览器中右键选择“查看源代码”或“查看网页源代码”，或其他最接近的选项），会看到每本书（或每部视频）都唯一地包含在一个表格单元格元素 <td> 中，它的类是 thumbtext。下面的内容是某本书相关的 HTML（一个删减的版本）：

```
<td class="thumbtext">
  <div class="thumbcontainer">
    <div class="thumbdiv">
      <a href="/product/9781118903407.do">
        <img src="..."/>
      </a>
    </div>
  </div>
  <div class="widthchange">
    <div class="thumbheader">
      <a href="/product/9781118903407.do">Getting a Big Data Job For Dummies</a>
    </div>
    <div class="AuthorName">By Jason Williamson</div>
    <span class="directorydate">        December 2014     </span>
    <div style="clear:both;">
      <div id="146350">
        <span class="pricelabel">
                            Ebook:

                            <span class="price"> $29.99</span>
        </span>
      </div>
    </div>
  </div>
</td>
```

良好的开端是找到所有的 td thumbtext 标签元素：

```
tds = soup('td', 'thumbtext')
print len(tds)
# 30
```

接下来我们要过滤掉视频。（那位潜在的投资者只对书感兴趣。）如果我们进一步地检查 HTML，会看到每个 td 会包含一个或更多个类为 pricelabel 的 span 元素，它的文本看起来像 Ebook: 或者 video: 或者 Print:。看起来视频仅包含一个 pricelabel，它的文本以 Video（在移除前导空格之后）开头。这意味着我们可以这样来检测视频：

```
def is_video(td):
    """it's a video if it has exactly one pricelabel, and if
    the stripped text inside that pricelabel starts with 'Video'"""
```

```
    pricelabels = td('span', 'pricelabel')
    return (len(pricelabels) == 1 and
            pricelabels[0].text.strip().startswith("Video"))

print len([td for td in tds if not is_video(td)])
# 对我来说结果是21,你得到的结果可能会不同
```

现在我们已准备好要从 td 元素中提取数据了。看起来图书的标题是包含在 <div class="thumbheader"> 里的标签 <a> 中的文本：

```
title = td.find("div", "thumbheader").a.text
```

作者（们）的名字在 AuthorName <div> 的文本里。它们由一个 By（我们打算去掉它）开头，由逗号分隔（我们打算把它们分隔开，然后去掉其中的空格）：

```
author_name = td.find('div', 'AuthorName').text
authors = [x.strip() for x in re.sub("^By ", "", author_name).split(",")]
```

ISBN 看起来是包含在 thumbheader <div> 中的链接里：

```
isbn_link = td.find("div", "thumbheader").a.get("href")

# re.match捕捉了括号中的正则表达式部分
isbn = re.match("/product/(.*)\.do", isbn_link).group(1)
```

日期就是 <span class="directorydate"> 的内容：

```
date = td.find("span", "directorydate").text.strip()
```

让我们把所有这些都放到一个函数里边：

```
def book_info(td):
    """given a BeautifulSoup <td> Tag representing a book,
    extract the book's details and return a dict"""

    title = td.find("div", "thumbheader").a.text
    by_author = td.find('div', 'AuthorName').text
    authors = [x.strip() for x in re.sub("^By ", "", by_author).split(",")]
    isbn_link = td.find("div", "thumbheader").a.get("href")
    isbn = re.match("/product/(.*)\.do", isbn_link).groups()[0]
    date = td.find("span", "directorydate").text.strip()

    return {
        "title" : title,
        "authors" : authors,
        "isbn" : isbn,
        "date" : date
    }
```

现在我们准备好进行抓取了：

```
from bs4 import BeautifulSoup
import requests
from time import sleep
base_url = "http://shop.oreilly.com/category/browse-subjects/" + \
           "data.do?sortby=publicationDate&page="

books = []

NUM_PAGES = 31     # 这是写作本书时的值,现在有可能更多

for page_num in range(1, NUM_PAGES + 1):
    print "souping page", page_num, ",", len(books), " found so far"
    url = base_url + str(page_num)
    soup = BeautifulSoup(requests.get(url).text, 'html5lib')

    for td in soup('td', 'thumbtext'):
        if not is_video(td):
            books.append(book_info(td))

# 现在做一个好公民,遵守robots.txt!
sleep(30)
```

像这样从 HTML 中提取数据更像是一种数据艺术而不是数据科学。除了上例之外，你还可以从 HTML 中实施查找图书、查找标题等不计其数的类似的逻辑行为。

既然已经收集好了数据，现在就可以把每一年出版的图书数据绘制出来（如图 9-1）：

```
def get_year(book):
    """book["date"] looks like 'November 2014' so we need to
    split on the space and then take the second piece"""
    return int(book["date"].split()[1])

# 2014是包含数据的最后一个完整的年份(我运行这段代码的时间)
year_counts = Counter(get_year(book) for book in books
                      if get_year(book) <= 2014)

import matplotlib.pyplot as plt
years = sorted(year_counts)
book_counts = [year_counts[year] for year in years]
plt.plot(years, book_counts)
plt.ylabel("数据图书的数量")
plt.title("数据大发展！")
plt.show()
```

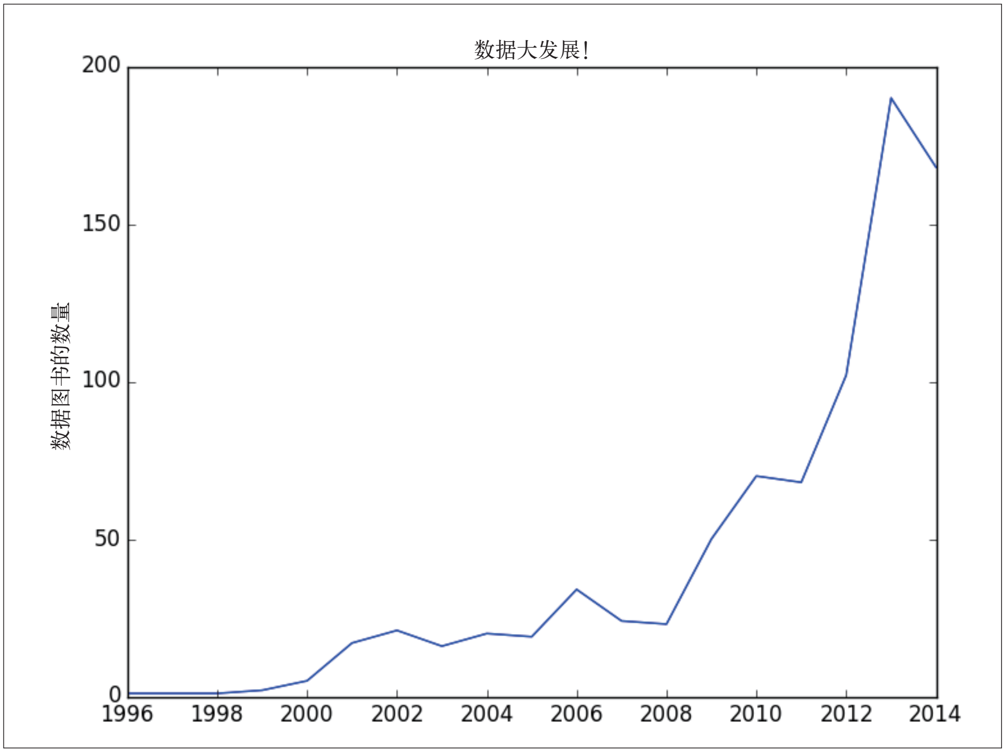


图 9-1：每年数据图书的出版数量

不幸的是，那位潜在的投资者看到这张图后断言 2013 年是“数据时代的巅峰”。

9.4　使用API

许多网站和网络服务提供相应的应用程序接口（Application Programming Interface，API），允许你明确地请求结构化格式的数据。这省去了你不得不抓取数据的麻烦！

9.4.1　JSON（和XML）

因为 HTTP 是一种转换文本的协议，你通过网络 API 请求的数据需要串行化（serialized）地转换为字符串格式。通常这种串行化使用 JavaScript 对象符号（JavaScript Object Notation，JSON）。JavaScript 对象看起来和 Python 的字典很像，使得字符串表达非常容易解释：

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2014,
  "topics" : [ "data", "science", "data science"] }
```

我们可以使用 Python 的 `json` 模块来解析 JSON。尤其是，我们会用到它的 `loads` 函数，这个函数可以把一个代表 JSON 对象的字符串反串行化（deserialize）为 Python 对象：

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2014,
                  "topics" : [ "data", "science", "data science"] }"""

# 解析JSON以创建一个Python字典
deserialized = json.loads(serialized)
if "data science" in deserialized["topics"]:
    print deserialized
```

有时候 API 的提供者可能会不耐烦，只给你提供 XML 格式的响应：

```
<Book>
  <Title>Data Science Book</Title>
  <Author>Joel Grus</Author>
  <PublicationYear>2014</PublicationYear>
  <Topics>
    <Topic>data</Topic>
    <Topic>science</Topic>
    <Topic>data science</Topic>
  </Topics>
</Book>
```

我们也可以仿照从 HTML 获取数据的方式，用 `BeautifulSoup` 从 XML 中获取数据；更多细节可查阅文档。

9.4.2 使用无验证的API

现在大多数的 API 要求你在使用之前先验证身份。而若我们不愿勉强自己屈就这种政策，API 会给出许多其他的陈词滥调来阻止我们的浏览。因此，先来看一下 GitHub 的 API（https://developer.github.com/v3/），利用它我们可以做一些简单的无需验证的事情：

```
import requests, json
endpoint = "https://api.github.com/users/joelgrus/repos"

repos = json.loads(requests.get(endpoint).text)
```

此处 `repos` 是一个 Python 字典的列表，其中每一个字典表示我的 GitHub 账户的一个代码仓库。（可以随意替换成你的用户名，以取得你的代码仓库的数据。你有 GitHub 账号，对吧？）

这个结果能指出哪个月哪一周的哪一天我最愿意创建代码仓库。唯一的问题是，响应里的日期是（Unicode）字符串：

```
u'created_at': u'2013-07-05T02:02:28Z'
```

Python 本身没有很强大的日期解析器，所以我们需要安装一个：

```
pip install python-dateutil
```

其中你需要的可能只是 dateutil.parser.parse 函数：

```
from dateutil.parser import parse

dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

类似地，你可以获取我最后五个代码仓库所用的语言：

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["created_at"],
                             reverse=True)[:5]

last_5_languages = [repo["language"]
                    for repo in last_5_repositories]
```

通常我们无需在“做出请求而且自己解析响应”这种低层次上使用 API。使用 Python 的好处之一是已经有人建好了库，方便你访问你感兴趣的几乎所有 API。这些库可以把事情做好，为你省下查找 API 访问的诸多冗长细节的麻烦。（如果这些库不能很好地完成任务，或者它们依赖的是对应的 API 已失效的版本，那就会给你带来巨大的麻烦。）

尽管如此，偶尔你还是需要操作你自己的 API 访问库（或者，更常见的，去调试别人不能顺利操作的库），所以了解一些细节是很有好处的。

9.4.3 寻找API

如果你需要一个特定网站的数据，可以查看它的开发者部分或 API 部分的细节，然后以关键词“python__api”在网络上搜索相应的库。Python 有一个 Rotten Tomatoes 的库。Python 还有针对 Klout、Yelp、IMDB 等的多个 API 封装。

如果你想查看有 Python 封装的 API 列表，可参阅 Python API（http://www.pythonapi.com/）和 Python for Beginners（http://www.pythonforbeginners.com/development/list-of-python-apis/）中的两个名录。

如果你想要的是一份更宽泛的网络 API 名录（不一定含有 Python 封装），Programmable Web（http://www.programmableweb.com/）是个好的资源，它有一个关于分好类的 API 的庞大名录。

如果最终还是找不到你需要的 API，还是可以通过抓取获得的。这是数据科学家最后的绝招。

9.5　案例：使用Twitter API

Twitter 是一个非常好的数据源。你可以从它得到实时的新闻，可以用它来度量对当前事件的反应，可以利用它找到与特定主题有关的链接。使用 Twitter 可以做几乎任何你能想到的事，只要你能获得它的数据。可以通过它的 API 来获得数据。

为了和 Twitter API 互动，我们需要使用 Twython 库（`pip install twython`，https://github.com/ryanmcgrath/twython）。实际上有很多 Python Twitter 的库，但这一个是我用过的库中最好用的一个。你也可以尝试一下其他的库。

获取证明文件

为了使用 Twitter 的 API，需要先获取一些证明文件（为此你无论如何都要有一个 Twitter 的账户，这样你就能成为一个活跃友好的 Twitter `#datascience` 社区的一部分）。就像那些所有我不能控制的网站的指令一样，它们会在某个时刻过时，但是现在还是能发挥一段时间的作用的。（尽管在我写作本书的这段时间里，它们至少已经变更过一次了，所以祝你好运！）

1. 找到链接 https://apps.twitter.com/。
2. 如果你还没有注册，点击“注册”，并输入你的 Twitter 用户名和密码。
3. 点击“创建新 App”。
4. 给它起个名字（比如“数据科学”）并添加一些描述，放上一个任意的 URL 作为网址（不用在乎是哪个）。
5. 同意“服务条款”并点击“创建”。
6. 注意消费者钥匙（consumer key）和消费者密码（consumer secret）。
7. 点击“创建我的访问令牌”（access token）。
8. 注意访问令牌和访问令牌密码（你可能需要刷新页面）。

消费者钥匙和消费者密码告诉 Twitter 什么应用正在访问它的 API，而访问令牌和访问令牌密码告诉 Twitter 是谁正在访问它的 API。如果你曾经用 Twitter 账户访问过一些其他网站，“点击验证”页面会生成一个访问令牌，网站会用这个令牌来告诉 Twitter 访问者是你（或者至少是来自你的操作）。因为不需要这种“让任何人登录”的功能，我们可以获得静态生成的访问令牌和访问令牌密码。

> 消费者钥匙 / 密码和访问令牌钥匙 / 密码应该被看成是密码（password）。你不该分享它们，不该把它们印在书里，也不应该把它们记录在 GitHub 公共代码库里。一种简单的方法是把它们存储在不会被签入的 credentials.json 文件里，而且可以使用 `json.loads` 取回它们。

使用Twython

首先我们来看看 Search API（https://dev.twitter.com/docs/api/1.1/get/search/tweets），这个操作只需要消费者钥匙和密码，无需访问令牌或密码：

```
from twython import Twython

twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)

# 搜索包含短语“数据科学”的推文
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"].encode('utf-8')
    text = status["text"].encode('utf-8')
    print user, ":", text
    print
```

> 因为推文中经常包含 `print` 函数无法处理的 Unicode 字符，所以有必要使用 `.encode("utf-8")` 来应对这个问题。（如果对这个问题放任不管，很有可能会得到 `UnicodeEncodeError` 的报错。）
>
> 几乎可以肯定，你会在数据科学家职业生涯的某些时刻遇到严重的 Unicode 问题，这时你需要参考 Python 文档（https://docs.python.org/2/howto/unicode.html）或者勉为其难地开始使用 Python 3 吧，因为它能更好地处理 Unicode 文本。

如果你运行这段代码，会得到一些像这样的推文：

```
haithemnyc: Data scientists with the technical savvy &amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization.  So easy and Awesome.
```

这并不十分有趣，很大程度上是因为 Twitter Search API 只给你显示它认为最近的结果，无论内容有多么少。但对于数据科学工作，通常你需要大量的推文。这时，Streaming API（https://dev.twitter.com/streaming/reference/get/statuses/sample）就有用武之地了，它允许你连接到强大的 Twitter firehose 接口（的一个样本）。你需要使用访问令牌进行验证，才可以使用这个 API。

为了用 Twython 访问 Streaming API，需要定义一个从 `TwythonStreamer` 继承的类，并用这个类的 `on_success` 方法覆盖（当然也可能是用它的 `on_error` 方法来覆盖）：

```
from twython import TwythonStreamer

# 把数据添加到全局变量是一种非常差的形式
```

```
# 但会让这个例子更简单
tweets = []

class MyStreamer(TwythonStreamer):
    """our own subclass of TwythonStreamer that specifies
    how to interact with the stream"""

    def on_success(self, data):
        """what do we do when twitter sends us data?
        here data will be a Python dict representing a tweet"""

        # 只收集英文的推文
        if data['lang'] == 'en':
            tweets.append(data)
            print "received tweet #", len(tweets)

        # 当收集了足够多的推文就停止
        if len(tweets) >= 1000:
            self.disconnect()

   def on_error(self, status_code, data):
       print status_code, data
       self.disconnect()
```

MyStreamer 会连接到 Twitter 流并等待 Twitter 给它发送数据。它每收到一些数据（在这里，一条推文表示为一个 Python 对象）就传递给 on_success 方法，如果推文是英文的，这个方法会把推文附加到 tweets 列表中，在收集到 1000 条推文后会断开和流的连接。

剩下的工作就是初始化和启动运行了：

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

# 开始使用包含关键词'data'的公共状态
stream.statuses.filter(track='data')

# 如果我们想使用*all*公共状态的样本
# stream.statuses.sample()
```

它会一直运行下去直到收集 1000 条推文为止（或直到遇到一个错误为止），此时就可以着手分析这些推文了。比如，你可以用下面的方法寻找最常见的标签：

```
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet["entities"]["hashtags"])

print top_hashtags.most_common(5)
```

每条推文都包含许多数据。你可以自己尝试一下各种方法，或仔细查阅 Twitter API 的文档（https://dev.twitter.com/overview/api/tweets）。

在一个正式的项目中，你可能并不想依赖内存中的列表来存储推文。相反，你可能想把推文保存在文件或者数据库中，这样就可以永久地拥有它们。

9.6 延伸学习

- pandas（http://pandas.pydata.org/）是数据科学用来处理（特别是导入）数据的一个主要的库。
- Scrapy（http://scrapy.org/）是一个特性更全的库，可用来构建更复杂的网络抓取器，来执行类似跟踪未知链接等任务。

第10章
数据工作

专家更依赖数据，而非主观判断。

——科林·鲍威尔

数据工作既是艺术又是科学。前面我们讨论的大多是数据的科学的一面，这一章我们来管窥其艺术的一面。

10.1　探索你的数据

当确定了需要研究的问题，并已获取了一些数据时，你摩拳擦掌地恨不得马上建模求解。但是，你需要克制一下。首先，你应该探索数据。

10.1.1　探索一维数据

最简单的情形是，你得到的一个数据集合仅仅是一维数据集。比如，它们可以是每个用户在你的网站上平均每天花费的时间，每个数据科学教程视频的观看次数，或者是你的数据科学图书馆中每本数据科学书的页数。

第一步显然是计算一些总结性统计数据。比如你可能想知道你的数据集中有多少个数据点，最小值是多少，最大值是多少，平均值是多少，或者标准差是多少。

如果你仍不能很好地理解以上步骤，那么下一步最好是绘出直方图，即将你的数据分组成离散的区间（bucket），并对落入每个区间的数据点进行计数：

```
def bucketize(point, bucket_size):
    """floor the point to the next lower multiple of bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points, bucket_size):
    """buckets the points and counts how many in each bucket"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points, bucket_size, title=""):
     histogram = make_histogram(points, bucket_size)
     plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
     plt.title(title)
     plt.show()
```

比如，考虑以下两个数据集：

```
random.seed(0)

# -100到100之间均匀抽取
uniform = [200 * random.random() - 100 for _ in range(10000)]

# 均值为0标准差为57的正态分布
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(10000)]
```

这两个数据集的均值都接近 0，标准差都接近 58 ，但它们的分布非常不同。图 10-1 展示了均匀分布。

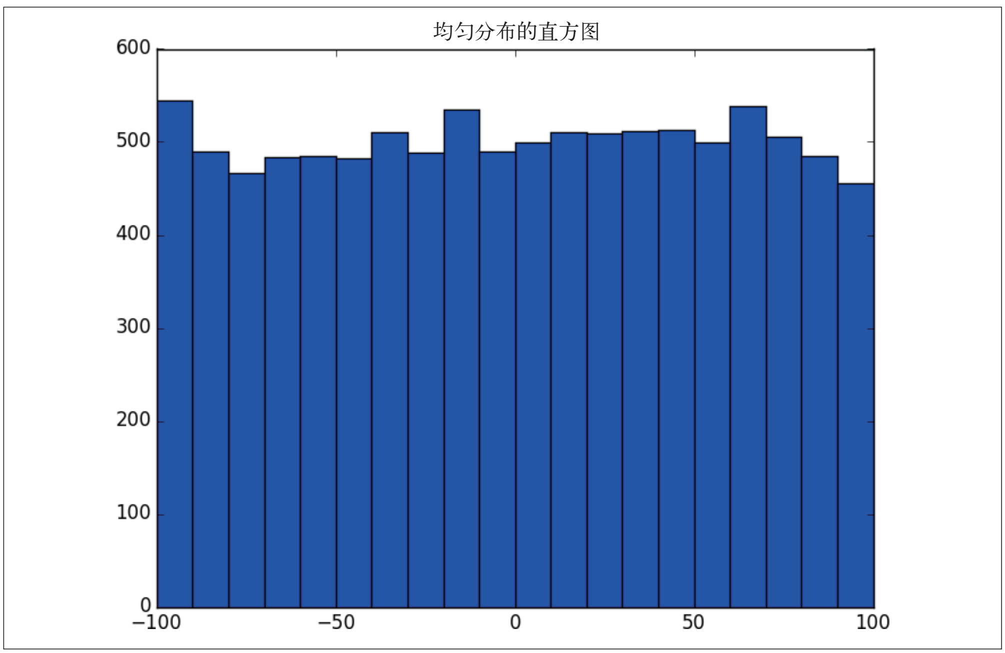


图 10-1：均匀分布的直方图

```
plot_histogram(uniform, 10, "均匀分布的直方图")
```

而图 10-2 展示了正态分布：

```
plot_histogram(normal, 10, "正态分布的直方图")
```

这两种分布有非常不同的最大值和最小值。但是，仅仅知道这一点并不足以理解它们有何差异。

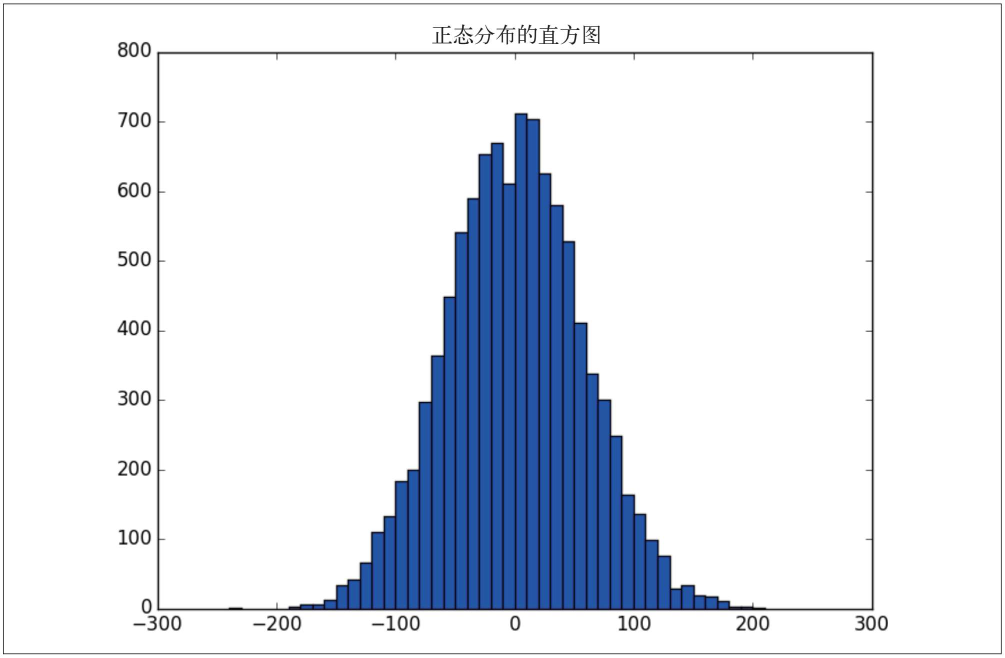


图 10-2：正态分布的直方图

10.1.2　二维数据

现在假设你的数据集是二维的。也许在每天上网时间之外还增加了数据科学工作年限。你当然会希望能从每个维度上单独理解数据，但也许你更希望综合两个维度来考察数据。

比如，考察下面一个伪数据集：

```
def random_normal():
    """returns a random draw from a standard normal distribution"""
    return inverse_normal_cdf(random.random())

xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

如果你对 ys1 和 ys2 运行 plot_histogram 程序，会得到很相似的直方图（事实上，两个正态分布的均值和标准差都相同）。

但是在联合分布上，每个都与 xs 有很大差别，如图 10-3 所示：

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray',  label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("差别很大的联合分布")
plt.show()
```

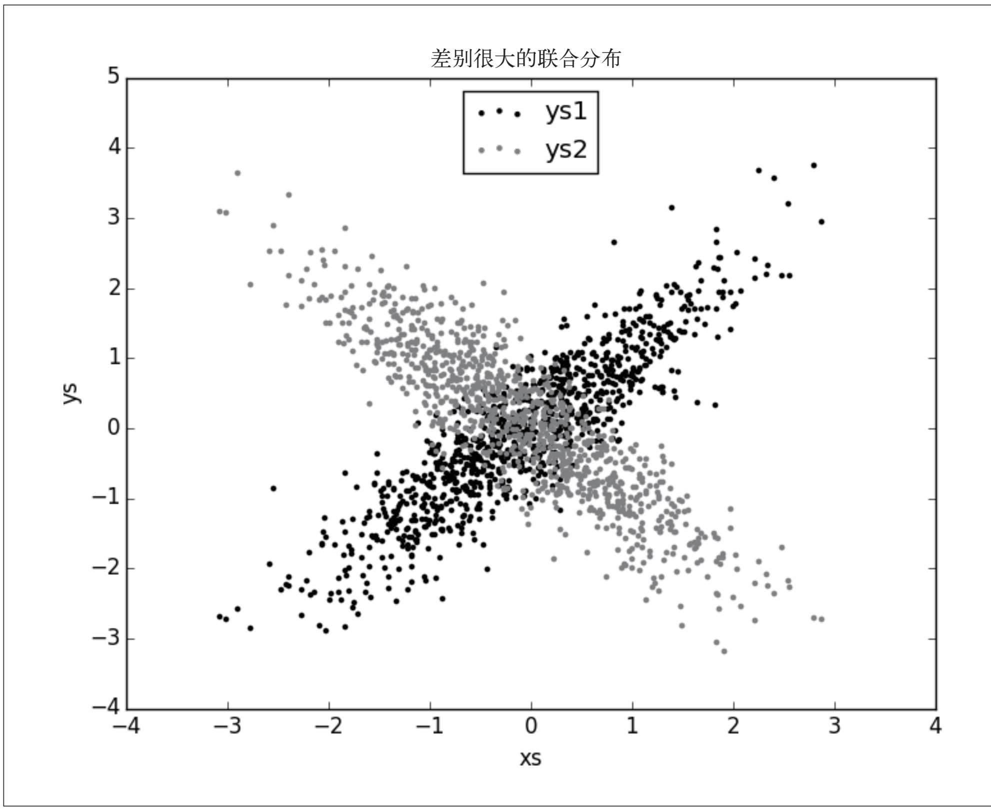


图 10-3：两个不同的 ys 的散点图

如果你考察相关性，差异会非常明显：

```
print correlation(xs, ys1)    #  0.9
print correlation(xs, ys2)    # -0.9
```

10.1.3 多维数据

对于多维数据，你可能想了解各个维度之间是如何相关的。一个简单的方法是考察相关矩阵（correlation matrix），矩阵中第 *i* 行第 *j* 列的元素表示第 *i* 维与第 *j* 维数据的相关性：

```
def correlation_matrix(data):
    """returns the num_columns x num_columns matrix whose (i, j)th entry
    is the correlation between columns i and j of data"""

    _, num_columns = shape(data)

   def matrix_entry(i, j):
       return correlation(get_column(data, i), get_column(data, j))

   return make_matrix(num_columns, num_columns, matrix_entry)
```

一个更为直观的方法（如果维度不太多）是做散点图矩阵（图 10-4），以展示配对散点图。通过命令 plt.subplots() 可以生成子图。我们给出了行数和列数，它返回一个 figure 对象（我们不会用到它）和一个 axes 对象的二维数组（每个都会绘出）：

```
import matplotlib.pyplot as plt

_, num_columns = shape(data)
fig, ax = plt.subplots(num_columns, num_columns)

for i in range(num_columns):
    for j in range(num_columns):

        # x轴上column_j对y轴上column_i的散点
        if i != j: ax[i][j].scatter(get_column(data, j), get_column(data, i))

        # 只有当 i == j时显示序列名
        else: ax[i][j].annotate("series " + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")

        # 除了图的左侧和下方之外,隐藏图的标记
        if i < num_columns - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

# 修复右下方和左上方的图标记
# 因为它们只有文本,是错误的
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())

plt.show()
```

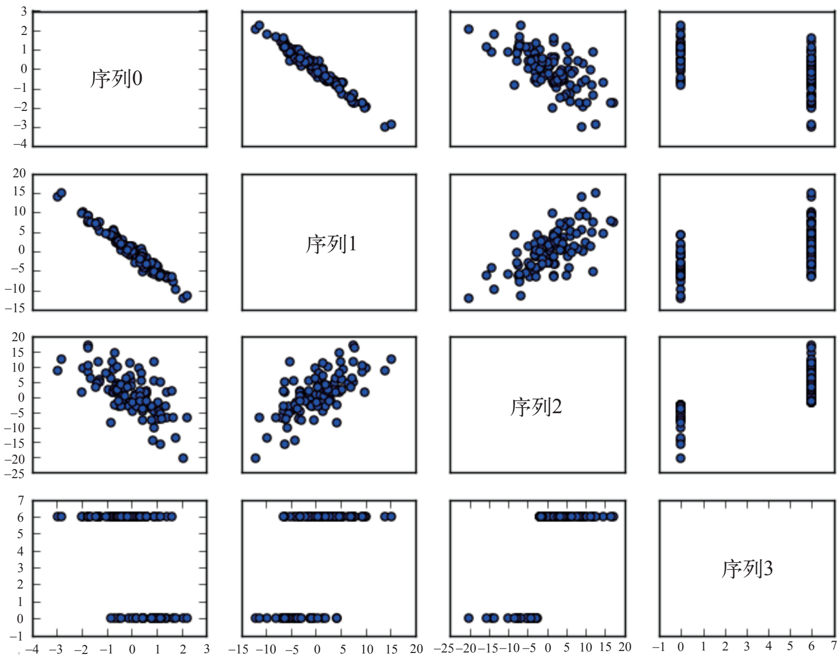


图 10-4：散点图矩阵

通过这些散点图你会看出，序列 1 与序列 0 的负相关程度很高，序列 2 和序列 1 的正相关程度很高，序列 3 的值仅有 0 和 6，并且 0 对应序列 2 中较小的值，6 对应较大的值。

这是一种能让你查看变量之间大概的相关度的快捷方法（除非你为了查看更加具体的效果而花费数小时调整 `matplotlib`，这样就不快捷了）。

10.2　清理与修改

真实世界的数据是有很多问题的。在使用数据之前，你通常需要对它们进行一定的预处理。我们在第 9 章举过这样的例子。我们需要把字符串转化成可以使用的浮点型数据（`float`）或者整型数据（`int`）。以前，我们在使用数据之前会这样做：

```
closing_price = float(row[2])
```

但通过建立包括 `csv.reader` 的函数，这样进行解析更不容易触发误差。我们会列出一系列解析器，每个解析器具体说明其中一列如何解析。我们会用 `None` 表示“对这列什么都不做”：

```
def parse_row(input_row, parsers):
    """given a list of parsers (some of which may be None)
    apply the appropriate one to each element of the input_row"""

    return [parser(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]

def parse_rows_with(reader, parsers):
    """wrap a reader to apply the parsers to each of its rows"""
    for row in reader:
        yield parse_row(row, parsers)
```

如果有不良数据怎么办？一个浮点值是否真正代表一个数字？我们通常会使用一个 None 函数而非硬跑程序。我们可以通过一个辅助函数来解决：

```
def try_or_none(f):
    """wraps f to return None if f raises an exception
    assumes f takes only one input"""
    def f_or_none(x):
        try: return f(x)
        except: return None
    return f_or_none
```

然后我们重写 parse_row 来使用它：

```
def parse_row(input_row, parsers):
    return [try_or_none(parser)(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]
```

比如，如果我们用逗号分割的股票数据中有不良数据：

```
6/20/2014,AAPL,90.91
6/20/2014,MSFT,41.68
6/20/3014,FB,64.5
6/19/2014,AAPL,91.86
6/19/2014,MSFT,n/a
6/19/2014,FB,64.34
```

我们现在可以在一个单独步骤中读入和解析：

```
import dateutil.parser
data = []

with open("comma_delimited_stock_prices.csv", "rb") as f:
    reader = csv.reader(f)
    for line in parse_rows_with(reader, [dateutil.parser.parse, None, float]):
        data.append(line)
```

然后我们只需检查其中 None 的行数：

```
for row in data:
    if any(x is None for x in row):
        print row
```

然后再决定如何处理它们。（一般来说，你有三个选择：删除它们；溯源并修复不良数据或缺失数据；什么都不做，自求多福吧。）

我们可以为 csv.DictReader 创建相似的帮助函数。这样的话，你很可能希望提供基于域名的解析字典。例如：

```
def try_parse_field(field_name, value, parser_dict):
    """try to parse value using the appropriate function from parser_dict"""
    parser = parser_dict.get(field_name)   # 如果没有此条目,则为None
    if parser is not None:
        return try_or_none(parser)(value)
    else:
        return value

def parse_dict(input_dict, parser_dict):
    return { field_name : try_parse_field(field_name, value, parser_dict)
             for field_name, value in input_dict.iteritems() }
```

接下来最好是使用 10.1 节所讲的技术或即时分析来确认异常值。比如，如果你发现股票文件中有一个数据的时间是 3014 年，这不会给你报错提示，但这显然是错误的数据。如果你没有发现这个错误，就会得到很糟糕的结果。真实世界的数据集充斥着诸如小数点缺失、多余的零、排印错误等无数各种各样的错误，找出错误是你责无旁贷的工作。（也许这不是你的正式工作，但这工作又非你做不可。）

10.3 数据处理

数据科学家的核心技能之一就是*处理数据*。与其说它是一种特定的技术，不如说它是一种通用的方法，所以这里我们只通过一些例子窥其一二。

假设我们需要处理如下股票价格字典：

```
data = [
    {'closing_price': 102.06,
     'date': datetime.datetime(2014, 8, 29, 0, 0),
     'symbol': 'AAPL'},
    # ...
]
```

我们可以从概念上将它们理解为行（就像在一张表中）。

我们开始对这些数据发问。在这个过程中，我们会不断关注做事所使用的模式，并抽象出一些工具以使数据的处理更容易些。

比如，如果我们想知道 AAPL 有史以来的最高收盘价，可以将这个工作分解成具体的步骤：

(1) 将数据限定在 AAPL 行上；

(2) 从每行提取收盘价 closing_price；
(3) 取价格中的最大值 max。

我们可以使用一个列表解析一次性完成这三个步骤：

```
max_aapl_price = max(row["closing_price"]
                     for row in data
                     if row["symbol"] == "AAPL")
```

更一般地，我们也许希望知道数据集中每只股票的最高收盘价。一个方法如下所示。

(1) 聚集起股票代码（symbol）相同的行。
(2) 在每组中，重复之前的工作：

```
# 按股票代码对行分组
by_symbol = defaultdict(list)
for row in data:
    by_symbol[row["symbol"]].append(row)

# 使用字典解析找到每个股票代码的最大值
max_price_by_symbol = { symbol : max(row["closing_price"]
                                     for row in grouped_rows)
                        for symbol, grouped_rows in by_symbol.iteritems() }
```

有一些模式已经存在。在以上两个例子中，我们需要在每个字典 dict 中提取出收盘价 closing_price。因而我们可以创建一个函数，以从字典中提取一个字段，并创建另一个函数，以从字典集合中提取出同样的字段：

```
def picker(field_name):
    """returns a function that picks a field out of a dict"""
    return lambda row: row[field_name]

def pluck(field_name, rows):
    """turn a list of dicts into the list of field_name values"""
    return map(picker(field_name), rows)
```

我们同样可以建立一个函数，通过 group 函数的结果把行分组，并选择性地对每组使用 value_transform 函数：

```
def group_by(grouper, rows, value_transform=None):
    # 键是分组情况的输出,值是行的列表
    grouped = defaultdict(list)
    for row in rows:
        grouped[grouper(row)].append(row)

    if value_transform is None:
        return grouped
    else:
        return { key : value_transform(rows)
                 for key, rows in grouped.iteritems() }
```

这使得我们可以更简单地再现先前的例子。比如：

```
max_price_by_symbol = group_by(picker("symbol"),
                               data,
                               lambda rows: max(pluck("closing_price", rows)))
```

现在我们可以问一些更复杂的问题，比如在我们的数据集中，单日百分比变动的最大值和最小值分别是什么。百分比变动的公式是 `price_today/price_yesterday - 1`（即今天的价格／昨天的价格 −1）。这意味着我们需要用某种方式将今天的价格和昨天的价格联系起来。一种方法是按照符号将价格分组，再在每组中：

(1) 按照日期排列价格；
(2) 通过命令 `zip` 得到配对价格（前一天的，今天的）；
(3) 将配对价格转换为新的“百分比变动”行。

我们首先写一个函数，来完成每一组内的工作：

```
def percent_price_change(yesterday, today):
    return today["closing_price"] / yesterday["closing_price"] - 1

def day_over_day_changes(grouped_rows):
    # 按日期对行排序
    ordered = sorted(grouped_rows, key=picker("date"))

    # 对偏移量应用zip函数得到连续两天的成对表示
    return [{ "symbol" : today["symbol"],
              "date" : today["date"],
              "change" : percent_price_change(yesterday, today) }
            for yesterday, today in zip(ordered, ordered[1:])]
```

然后我们可以将它作为 `value_transform` 在 `group_by` 中使用：

```
# 键是股票代码,值是一个"change"字典的列表
changes_by_symbol = group_by(picker("symbol"), data, day_over_day_changes)

# 收集所有"change"字典放入一个大列表中
all_changes = [change
               for changes in changes_by_symbol.values()
               for change in changes]
```

在这个点上，很容易找到最大值与最小值：

```
max(all_changes, key=picker("change"))
# {'change': 0.3283582089552237,
#  'date': datetime.datetime(1997, 8, 6, 0, 0),
#  'symbol': 'AAPL'}
# see, e.g. http://news.cnet.com/2100-1001-202143.html
min(all_changes, key=picker("change"))
# {'change': -0.5193370165745856,
#  'date': datetime.datetime(2000, 9, 29, 0, 0),
```

```
#  'symbol': 'AAPL'}
# see, e.g. http://money.cnn.com/2000/09/29/markets/techwrap/
```

现在我们可以使用这个新的 all_changes 数据集来找出投资科技股的最佳月份。首先按月份对变化分组；然后在每组中计算整体变化。

我们再次写一个恰当的 value_transform 函数，然后使用 group_by 函数：

```
# 为了组合百分比的变化,我们对每一项加1,把它们相乘,再减去1
# 比如,如果我们组合 +10% 和 -20%，总体的改变是
#    (1 + 10%) * (1 - 20%) - 1 = 1.1 * .8 - 1 = -12%
def combine_pct_changes(pct_change1, pct_change2):
    return (1 + pct_change1) * (1 + pct_change2) - 1

def overall_change(changes):
    return reduce(combine_pct_changes, pluck("change", changes))

overall_change_by_month = group_by(lambda row: row['date'].month,
                                   all_changes,
                                   overall_change)
```

类似这样的数据处理方式将会贯穿全书，但它常常不会明显地引起我们的注意。

10.4　数据调整

许多技术对数据单位（scale）敏感。比如，假设你有一个包括数百名数据科学家的身高和体重的数据集，并且需要创建体型大小的聚类（cluster）。

直观上讲，我们用聚类表示相近的点，这意味着我们需要某种点距离的概念。我们知道有欧几里得距离函数 distance，所以自然地，一种方法是将数据对 (height, weight) 视为二维空间中的点。考虑表 10-1 中列出的观测对象。

表10-1：身高和体重

观测对象	身高（英寸）	身高（厘米）	体重（磅）
A	63	160	150
B	67	170.2	160
C	70	177.8	171

如果我们用英寸作为身高的单位，那么 B 最近的邻居是 A：

```
a_to_b = distance([63, 150], [67, 160])      # 10.77
a_to_c = distance([63, 150], [70, 171])      # 22.14
b_to_c = distance([67, 160], [70, 171])      # 11.40
```

但是，如果用厘米作为单位，那么 B 最近的邻居变成了 C：

```
a_to_b = distance([160, 150], [170.2, 160])   # 14.28
a_to_c = distance([160, 150], [177.8, 171])   # 27.53
b_to_c = distance([170.2, 160], [177.8, 171]) # 13.37
```

显然，如果单位变化导致结果发生这样的变化，那肯定是有问题的。因此，如果不同的维度之间不可比较，就需要对数据进行*调整*（rescale），以使得每个维度的均值为 0，标准差为 1。这种转换有效地摆脱了单位带来的问题，将每个维度转化为“与均值的标准差”。

首先，我们需要对每列计算均值和标准差：

```
def scale(data_matrix):
    """returns the means and standard deviations of each column"""
    num_rows, num_cols = shape(data_matrix)
    means = [mean(get_column(data_matrix,j))
             for j in range(num_cols)]
    stdevs = [standard_deviation(get_column(data_matrix,j))
              for j in range(num_cols)]
    return means, stdevs
```

然后用结果创建新的数据矩阵：

```
def rescale(data_matrix):
    """rescales the input data so that each column
    has mean 0 and standard deviation 1
    leaves alone columns with no deviation"""
    means, stdevs = scale(data_matrix)

    def rescaled(i, j):
        if stdevs[j] > 0:
            return (data_matrix[i][j] - means[j]) / stdevs[j]
        else:
            return data_matrix[i][j]

    num_rows, num_cols = shape(data_matrix)
    return make_matrix(num_rows, num_cols, rescaled)
```

一如既往地，你需要运用你的判断力。如果你拿到一个由身高和体重组成的巨大的数据集，需要将其过滤为仅由身高在 69.5 英寸至 70.5 英寸之间的人组成。很有可能（取决于你希望回答的问题）剩余的变差仅仅是*噪声*（noise），但你也许并不希望将其标准差与其他维度的标准差等而视之。

10.5 降维

有时候，数据的“真实”（或有用的）维度与我们掌握的数据维度并不相符。比如，考虑图 10-5 中所示的数据。

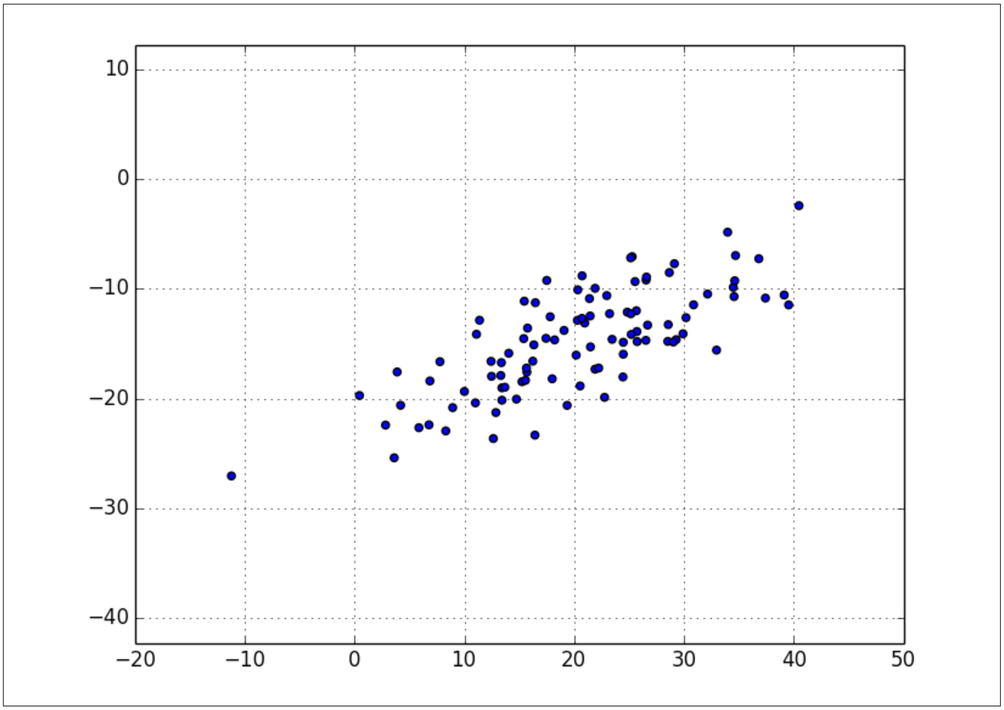


图 10-5：坐标轴“错误”的数据

数据的大部分变差看起来像是沿着单个维度分布的，既不与 *x* 轴对应，也不与 *y* 轴对应。

当这种情形发生时，我们可以使用一种叫作主成分分析（principal component analysis，PCA）的技术从数据中提取出一个或多个维度，以捕获数据中尽可能多的变差。

> 实际上，这样的技术不适用于低维数据集。降维多用于数据集的维数很高的情形，你可以通过一个小子集来抓住数据集本身的大部分变差。不过，这种情况很复杂，绝非一两章能讲得清。

首先，我们需要将数据转换成为每个维度均值为零的形式：

```
def de_mean_matrix(A):
    """returns the result of subtracting from every value in A the mean
    value of its column. the resulting matrix has mean 0 in every column"""
    nr, nc = shape(A)
    column_means, _ = scale(A)
    return make_matrix(nr, nc, lambda i, j: A[i][j] - column_means[j])
```

（如果不这样做，应用这种技术的结果可能就只是确定数据的均值本身，而非找出数据中的变差。）

图 10-6 展示了去均值后的示例数据。

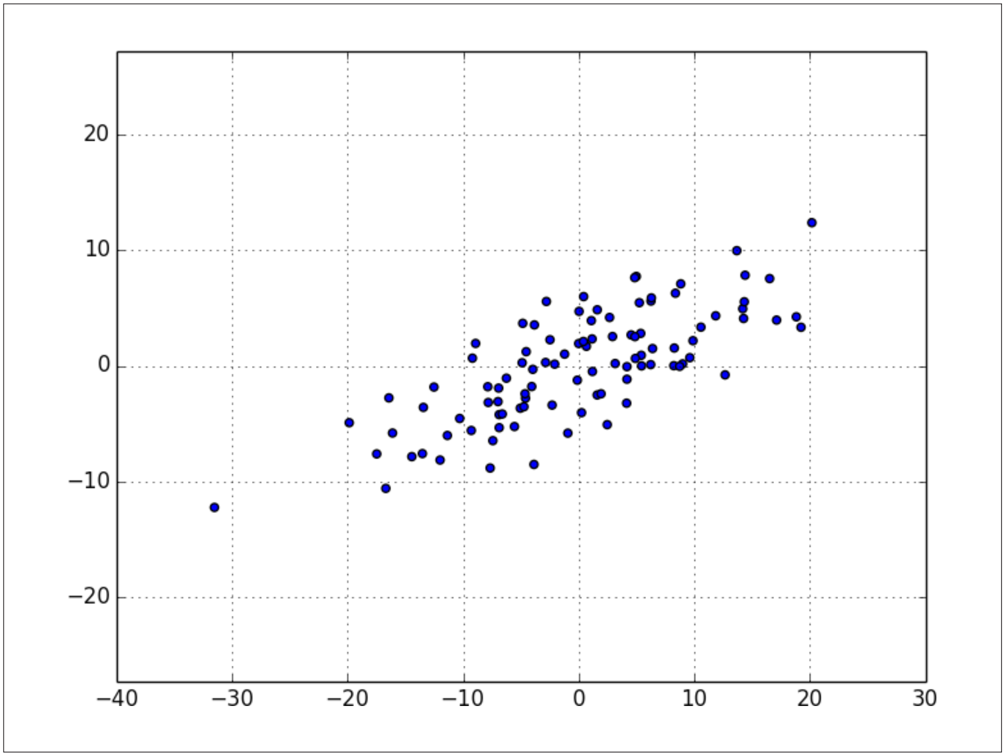


图 10-6：去均值后的数据

现在，已有一个去均值的矩阵 X，我们想问，最能抓住数据最大变差的方向是什么？

具体来说，给定一个方向 d（一个绝对值为 1 的向量），矩阵的每行 x 在方向 d 的扩展是点积 dot(x, d)。并且如果将每个非零向量 w 的绝对值大小调整为 1，则它们每个都决定了一个方向：

```
def direction(w):
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

因此，已知一个非零向量 w，我们可以计算 w 方向上的方差：

```
def directional_variance_i(x_i, w):
    """the variance of the row x_i in the direction determined by w"""
    return dot(x_i, direction(w)) ** 2

def directional_variance(X, w):
    """the variance of the data in the direction determined w"""
    return sum(directional_variance_i(x_i, w)
               for x_i in X)
```

我们可以找出使方差最大的那个方向。只要得到梯度函数，我们就可以通过梯度下降法计算出来：

```
def directional_variance_gradient_i(x_i, w):
    """the contribution of row x_i to the gradient of
    the direction-w variance"""
    projection_length = dot(x_i, direction(w))
    return [2 * projection_length * x_ij for x_ij in x_i]

def directional_variance_gradient(X, w):
    return vector_sum(directional_variance_gradient_i(x_i,w)
                      for x_i in X)
```

第一主成分仅是使函数 `directional_variance` 最大化的方向：

```
def first_principal_component(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_batch(
        partial(directional_variance, X),           # 现在是w的一个函数
        partial(directional_variance_gradient, X),  # 现在是w的一个函数
        guess)
    return direction(unscaled_maximizer)
```

也许，你也有可能使用随机梯度下降方法：

```
# 这里没有"y",所以我们仅仅是传递一个Nones的向量
# 和忽略这个输入的函数
def first_principal_component_sgd(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_stochastic(
        lambda x, _, w: directional_variance_i(x, w),
        lambda x, _, w: directional_variance_gradient_i(x, w),
        X,
        [None for _ in X],   # 假的 "y"
        guess)
    return direction(unscaled_maximizer)
```

对去均值的数据集，计算结果返回了方向 `[0.924, 0.383]`，这个方向看起来捕获了数据变动的主要方向轴（图 10-7）。

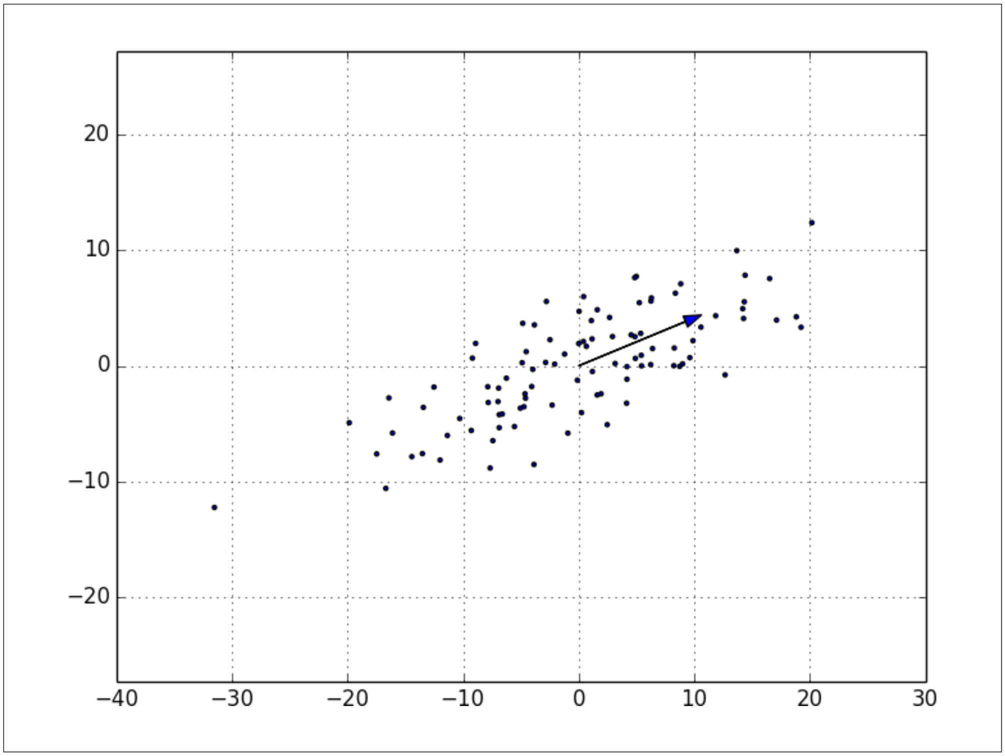


图 10-7：第一主成分

一旦我们找到了第一主成分的方向，就可以将数据在这个方向上投影得到这个成分的值：

```
def project(v, w):
    """return the projection of v onto the direction w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

如果还想得到其他的成分，就要先从数据中移除投影：

```
def remove_projection_from_vector(v, w):
    """projects v onto w and subtracts the result from v"""
    return vector_subtract(v, project(v, w))

def remove_projection(X, w):
    """for each row of X
    projects the row onto w, and subtracts the result from the row"""
    return [remove_projection_from_vector(x_i, w) for x_i in X]
```

因为这个例子中的数据集仅仅设定为二维，当移除第一主成分之后，剩下的实际上就是一个一维的成分了（图 10-8）。

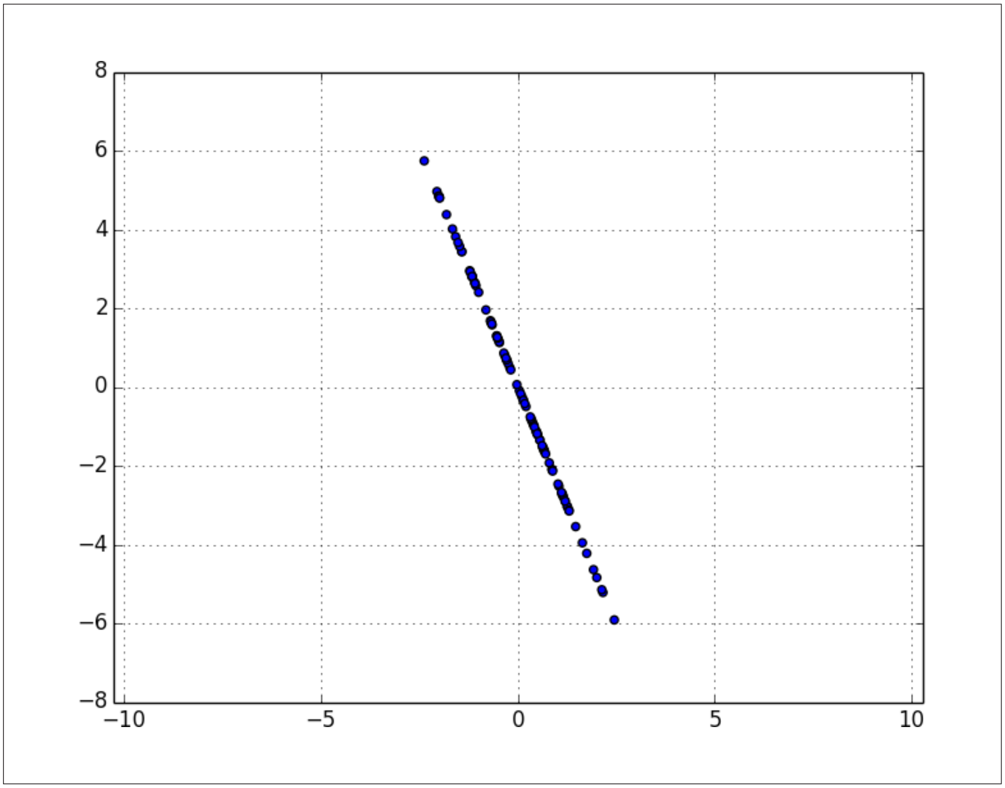


图 10-8：移除第一主成分之后的数据

在这点上，我们可以通过对 remove_projection 的结果重复这个过程来找到其他的主成分（图 10-9）。

在更高维的数据集中，我们可以通过迭代找到我们所需的任意数目的主成分：

```
def principal_component_analysis(X, num_components):
    components = []
    for _ in range(num_components):
        component = first_principal_component(X)
        components.append(component)
        X = remove_projection(X, component)

    return components
```

然后再将原数据转换为由主成分生成的低维空间中的点：

```
def transform_vector(v, components):
    return [dot(v, w) for w in components]

def transform(X, components):
    return [transform_vector(x_i, components) for x_i in X]
```

这种技术很有价值，原因有以下几点。首先，它可以通过清除噪声维度和整合高度相关的维度来帮助我们清理数据。

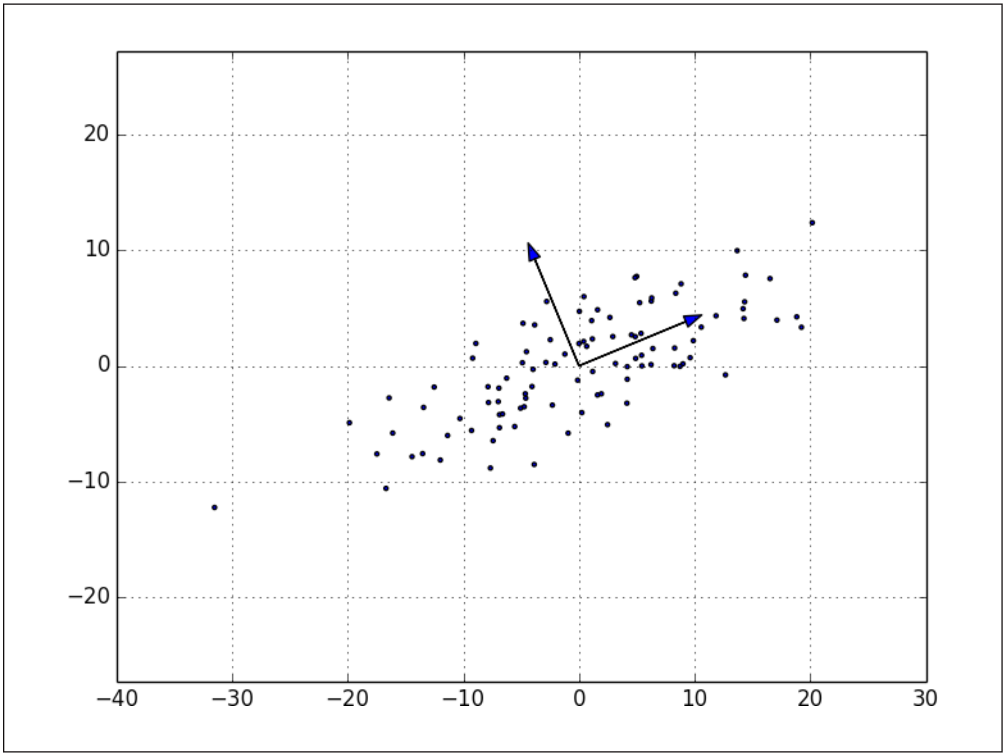


图 10-9：前两个主成分

第二，在提取出数据的低维代表后，我们就可以运用一系列并不太适用于高维数据的技术。我们可以在本书的很多地方看到运用这种技术的例子。

同时，它既可以帮助你建立更棒的模型，又会使你的模型更难理解。很容易理解诸如“工作年限每增加一年，平均工资会增加 1 万美元”这样的结论。但诸如“第三主成分每增加 0.1，平均工资就会增加 1 万美元”这样的结论就很难理解了。

10.6　延伸学习

- 正如我们在第 9 章末所提到的，pandas（http://pandas.pydata.org/）很可能是 Python 清理、整理、处理和利用数据的主要工具。本章中所有自己动手创建的范例都可以通过 pandas 更简单地完成。*Python for Data Analysis*（O’Reilly）大概是学习 pandas 的最好途径。
- scikit-learn 有多种多样的矩阵分解函数（http://scikit-learn.org/stable/modules/classes.html#module-sklearn.decomposition），包括 PCA。

第11章

机器学习

耳提面命诚可贵，求知若渴价更高。

——温斯顿·丘吉尔

在很多人眼里，数据科学几乎就是机器学习，而数据科学家每天做的事就是建立、训练和调整机器学习模型（而且，这些人当中有很大一部分并不真正知道机器学习是什么）。事实上，数据科学的主要内容是把商业问题转换为数据问题，然后收集数据、理解数据、清理数据、整理数据格式，而后才轮到机器学习这一后续工作。尽管如此，机器学习也是一种有趣且必要的后续工作，为了做好数据科学工作，你很有必要学习它。

11.1　建模

在讨论机器学习之前，我们需要谈谈模型（model）。

什么是模型？它实际上是针对存在于不同变量之间的数学（或概率）联系的一种规范。

比如，如果你想为你的社交网站融资，可以建立一个商业模型（大多数情况下建立在一个工作表里），模型的输入是诸如“用户数”“每位用户的广告收入”“雇员数”之类的变量，输出是接下来几年的年度利润。某本烹调指南涉及的模型是输入“吃饭的人数”和“饥饿的程度”来量化所需要的材料。如果你在电视上看过扑克比赛，就会知道选手们通过一个记牌的模型来实时地估计每位玩家的“获胜概率”，这个模型考虑了已经出的牌和还在牌桌上的牌的分布。

商业模型很可能是建立在简单的数学联系上：利润是收入减去支出，收入是平均价格乘以

单位销售量，诸如此类。菜谱模型则可能基于反复试验之上——某人走进厨房，尝试不同的原料组合，直到发现自己喜欢的口味。扑克模型基于概率论、扑克规则和某些关于处理牌的随机过程的合理假设。

11.2 什么是机器学习

关于什么是*机器学习*，每个人都有自己确切的定义。在这里，我们使用的定义是创建并使用那些由*学习数据*而得出的模型。在其他语境中，这也可以叫作*预测建模*或者*数据挖掘*，但是我们选择使用机器学习这个术语。一般来说，我们的目标是用已存在的数据来开发可用来对新数据*预测*多种可能结果的模型，比如：

- 预测一封邮件是否是垃圾邮件
- 预测一笔信用卡交易是否是欺诈行为
- 预测哪种广告最有可能被购物者点击
- 预测哪支橄榄球队会赢得超级杯大赛

下面我们会看到*有监督*的模型（其中的数据标注有正确答案，可供学习）和*无监督*的模型（没有标注）。还有一些其他类型的模型，我们不会在本书中进行讨论，如*半监督*的模型（其中有一部分数据带有标注）和*在线*的模型（模型需要根据新加入的数据做持续调整）。

现在，甚至在最简单的情况下都有一整套模型来描述我们感兴趣的联系。大多数情况下我们会自己选择*参数化*的模型族，然后使用数据来学习从某种程度上进行优化了的参数。

例如，我们假设一个人的身高（大致上）是他体重的线性函数，然后用数据来学习这个线性函数。或者，我们也可以假设决策树是一种用来诊断患者疾病的好方法，然后使用数据来学习一颗“最优”的树。本书剩余部分会穿插讲述可供我们学习的不同的模型族。

但在此之前，我们需要更好地理解机器学习的基础。本章剩余部分将探讨一些机器学习的基本概念，随后才会讨论到模型本身。

11.3 过拟合和欠拟合

在机器学习中，一种常见的困境是*过拟合*（overfitting）——指一个在训练数据上表现良好，但对任何新数据的泛化能力却很差的模型。这可能牵扯到对数据中*噪声*的学习，也可能涉及学习识别特别的输入，而不是对可以得到期望的输出进行准确预测的任何因素。

另一种有害的情况是*欠拟合*（underfitting），它产生的模型甚至在训练数据上都没有好的表现，尽管通常这暗示你模型不够好而要继续寻找改进的模型。

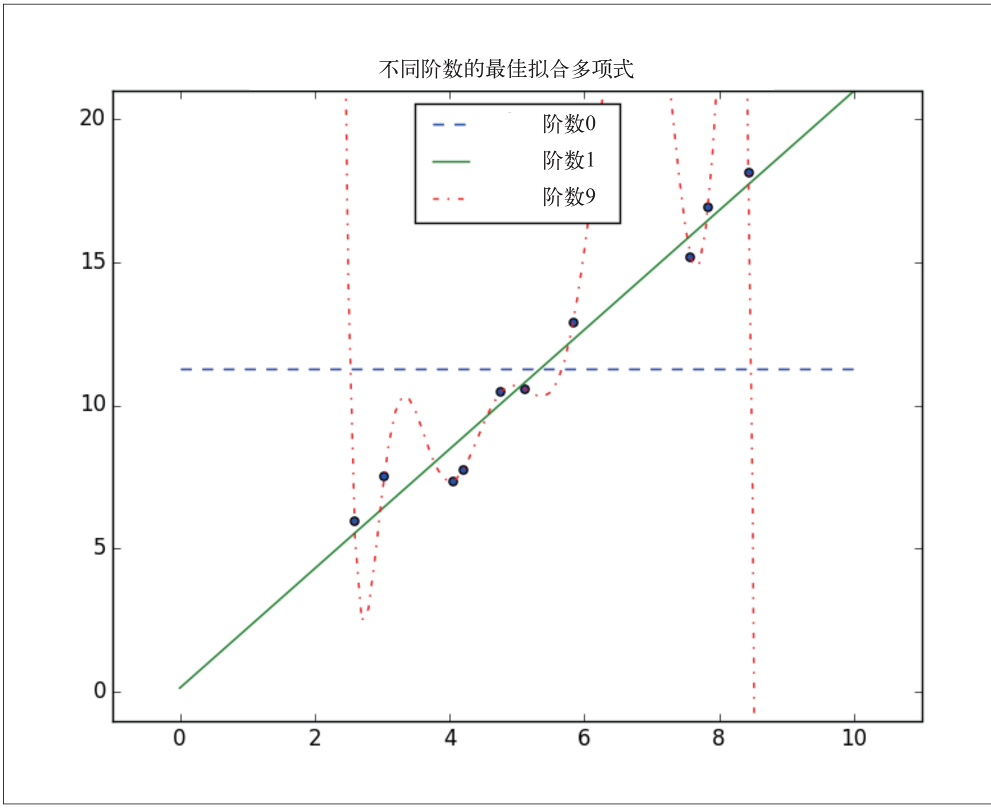


图 11-1：过拟合和欠拟合

在图 11-1 中，我对一组简单的数据拟合了 3 种多项式（别担心这是怎么做到的，我们会在后面的章节中介绍）。

水平线显示了最佳拟合阶数为 0（也就是常数）的多项式，它对训练数据来说存在严重的欠拟合。最佳拟合的阶数为 9（也就是有 10 个参数）的多项式精确地穿过训练数据的每个点，但这是严重过拟合的。如果我们能取到更多的一些点，这个多项式很有可能会偏离它们很多。阶数为 1 的线把握了很好的平衡——它和每个点都很接近，并且（如果这些数据是有代表性的）它也会和新的数据点很接近。

很明显，太复杂的模型会导致过拟合，并且在训练数据集之外不能很好地泛化。所以，我们该如何确保我们的模型不会太复杂呢？最基本的方法包括使用不同的数据来训练和测试模型。

最简单的做法是划分数据集，使得（比如说）三分之二的数据用来训练模型，之后用剩余的三分之一来衡量模型的表现：

```
def split_data(data, prob):
    """split data into fractions [prob, 1 - prob]"""
    results = [], []
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results
```

通常我们会有一个作为输入变量的矩阵 x 和一个作为输出变量的向量 y。这种情况下，我们要确保无论在训练数据还是测试数据中，都要把对应的值放在一起：

```
def train_test_split(x, y, test_pct):
    data = zip(x, y)                              # 成对的对应值
    train, test = split_data(data, 1 - test_pct)  # 划分这个成对的数据集
    x_train, y_train = zip(*train)                # 魔法般的解压技巧
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test
```

这样你就可以做一些类似下面这样的处理：

```
model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

如果模型对训练数据是过拟合的，那么它在（完全划分开的）测试数据集上会有可能真的表现得很不好，换句话说，如果它在测试数据上表现良好，那么你可以肯定地说它拟合良好而非过拟合。

然而在有些情况下这也可能会出错。

第一种情况是训练和测试数据集中的共有模式不能泛化到大型数据集上。

比如，假设数据集包括用户活跃度，每位用户每周一列。在此情形下，大多数用户会出现在训练数据和测试数据中，而且有些模型可能会学习识别用户而不是去发现涉及属性的联系。这不是个太大的问题，尽管我曾经遇到过一次。

一个更大的问题是，如果你划分训练集和测试集的目的不仅仅是为了判断模型，也是为了在许多模型中进行选择。此时，尽管不是所有的模型都是过拟合的，但“选择在测试集上表现最好的模型”是一种元训练（meta-training），会把测试集当作另一个训练集而运作。（当然，在测试集上有最好表现的模型会在测试集上有持续的好表现。）

在这种情况下，你应该把数据划分为三部分：一个用来建立模型的训练集，一个为在训练好的模型上进行选择的验证集，一个用来判断最终的模型的测试集。

11.4 正确性

当我不做数据科学的时候，我会涉猎医疗研究。在业余时间，我做了一个低成本、无害的新生儿测试——准确性高达 98%——测试新生儿是否会得白血病。我的律师确信我是有专利权的。所以我在这里把细节分享一下：仅仅当婴儿起名为 Luke 时预测会得白血病（因为这个名字听起来像白血病的英文 leukemia）。

如我们下面所见，这种测试确实有超过 98% 的准确性。然而，这是一个极为愚蠢的测试，也很好地解释了我们为什么不能仅用"准确率"这个概念来测量一个模型的好坏。

假设建立一个模型来做*二元*的判断，比如：一封邮件是否是垃圾邮件？我们是否该聘用这位应聘者？这个乘客是潜藏的恐怖分子吗？

给定一个标签数据集和一个预测模型，每个数据集都会落在下面其中一个属性中。

- 真阳性："这封邮件是垃圾邮件，我们做了正确的预测。"
- 假阳性（又称第 1 类错误）："这封邮件不是垃圾邮件，但是我们预测它是垃圾邮件。"
- 假阴性（又称第 2 类错误）："这封邮件是垃圾邮件，但是我们预测它不是垃圾邮件。"
- 真阴性："这封邮件不是垃圾邮件，而且我们正确地预测了它不是垃圾邮件。"

我们常用*混淆矩阵*（confusion matrix）中的计数来表示上面的四种情况：

	垃圾邮件	非垃圾邮件
预测"是垃圾邮件"	真阳性	假阳性
预测"非垃圾邮件"	假阴性	真阴性

让我们来看看我的白血病测试是如何符合这种框架的。现如今，大约每 1000 名婴儿中有 5 人会起名叫 Luke（http://www.babycenter.com/babyNameAllPops.htm?babyNameId=2918）。每人一生中罹患白血病的概率大约是 1.4%，或者说每 1000 人中会有 14 人患病（http://seer.cancer.gov/statfacts/html/leuks.html）。

如果我们相信这两个因素是独立的，然后对 1 百万人运用我的"Luke 是白血病患者"测试，预计能看到这样的混淆矩阵：

	白血病	非白血病	总计
Luke	70	4930	5000
非 Luke	13 930	981 070	995 000
总计	14 000	986 000	1 000 000

然后我们由此计算关于模型表现的多个统计量。例如，*准确率*（accuracy）定义为正确预测的比例：

```
def accuracy(tp, fp, fn, tn):
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

print accuracy(70, 4930, 13930, 981070)    # 0.98114
```

看起来这个数字令人印象十分深刻，但很明显这并不是一个好的测试，这意味着我们不能对原始的准确率有过多的信心。

更常见的做法是把查准率（precision）和查全率（recall）结合起来看待。查准率度量我的模型所做的关于“阳性”的预测有多准确：

```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

print precision(70, 4930, 13930, 981070)    # 0.014
```

查全率度量我的模型所识别的“阳性”的比例：

```
def recall(tp, fp, fn, tn):
    return tp / (tp + fn)

print recall(70, 4930, 13930, 981070)      # 0.005
```

这两个结果都低得可怕，反映出这是一个很不好的模型。

有时候可以把查准率和查全率组合成 F1 得分（F1 score），它是这样定义的：

```
def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

它是查准率和查全率的调和平均值（https://en.wikipedia.org/wiki/Harmonic_mean），因此必然会落在两者之间。

模型的选择通常是查准率和查全率之间的权衡。一个模型如果在信心不足的情况下预测“是”，那么它的查全率可能会较高，但查准率却较低；而如果一个模型在信心十足的情况下预测“是”，那么它的查全率可能会较低，但查准率却较高。

另一方面，也可以把这当作假阳性和假阴性之间的权衡。预测的“是”太多通常会给出很多的假阳性。预测的“否”太多通常会给出很多的假阴性。

假设对白血病来说有 10 个风险因素，你的身体具备的因素越多，就越容易患上白血病。这种情况下，你可以假设进行一系列连续性的测试：“至少有一个风险因素预测会得白血病”“至少有两个风险因素预测会得白血病”诸如此类。随着临界值的不断提高，测试的

查准率也提高了（因为具有更多风险因素的人更容易患上白血病），并且降低了测试的查全率（因为能够达到临界值的最终患病者越来越少）。对于类似这样的情况，选择合适的临界值实际上就是做出正确的权衡。

11.5 偏倚–方差权衡

思考过拟合问题的另一种角度是把它作为偏倚和方差之间的权衡。 偏倚和方差这两个名词是用来度量在（来自同一个大型总体的）不同的训练数据集上多次重复训练模型的情况。

比如，在 11.3 节“过拟合和欠拟合”中提到的 0 阶模型对（取自同一总体的）任何可能的训练集都会造成大量的错误。这表明该模型*偏倚*较高。然而任何两个随机选择的训练集会给出很相似的模型（因为任何两个随机选择的训练集都应该有大致相似的平均值）。所以我们称这个模型有低*方差*。高偏倚和低方差典型地对应着欠拟合。

另一方面，9 阶模型完美地拟合训练集，它具有很低的偏倚和很高的方差（因为任何两个训练集都可能给出非常不同的模型形式）。这种情况对应过拟合。

如果你的模型有高偏倚（这意味着即使在训练数据上也表现不好），可以尝试加入更多的特征。从 0 阶模型到 1 阶模型就是一个很大的改进。

如果你的模型有高方差，那可以类似地*移除*特征；另一种解决方法是（如果可能的话）获得更多的数据。

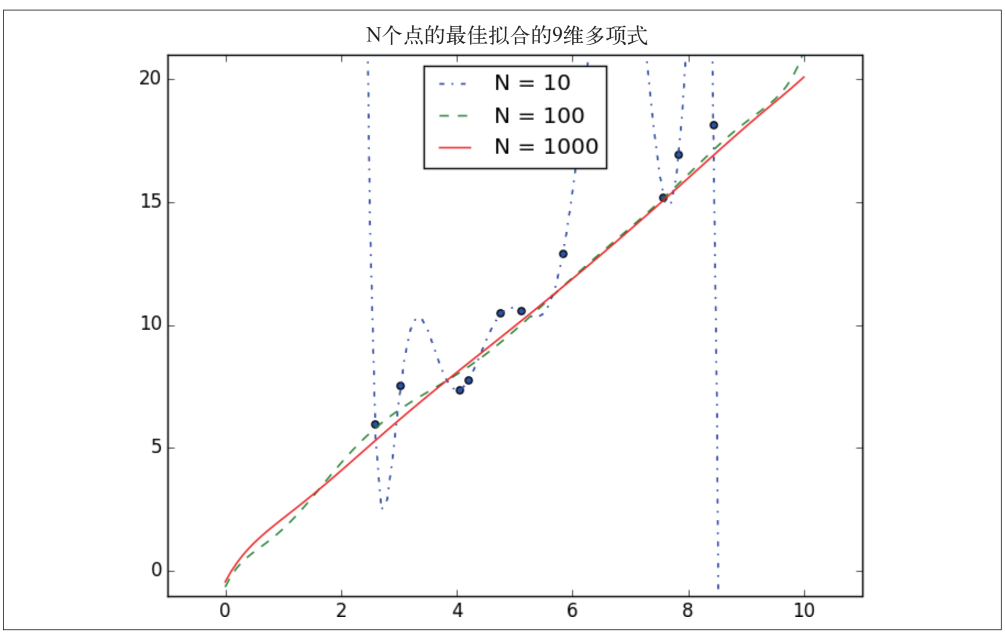


图 11-2：利用更多数据降低方差

在图 11-2 中，我们对不同大小的样本拟合了 9 阶多项式。如我们前面所见，基于 10 个点拟合的模型是一塌糊涂的。如果在 100 个数据点上训练，就会大大减少过拟合的问题。如果在 1000 个点上训练，看起来就像是 1 阶模型。

在模型复杂度不变的前提下，你有越多的数据，就越难过拟合。

另一方面，更多的数据对偏倚并不会有帮助。如果模型不能使用足够多的特征来捕捉数据的正则性，那么再多的数据也不会有帮助。

11.6 特征提取和选择

我们之前提到，如果数据没有足够的特征，模型很可能就会欠拟合。但如果数据有太多的特征，模型又容易过拟合。那什么是特征呢，它们又从何而来？

特征（feature）是指提供给模型的任何输入。在最简单的情况下，特征是直接提供给你的。如果你想基于某人的工作年限来预测其薪水，那工作年限就是你所拥有的唯一的特征。

（尽管如此，如同我们在 11.3 节“过拟合和欠拟合”中所见的，如果可以帮助你建立更好的模型，应该加入工作年限的平方项和立方项。）

当数据变得更复杂时，事情变得有趣起来。设想我们尝试建立一个垃圾邮件过滤器来预测一封邮件是否是垃圾邮件。大多数模型不知道如何处理原始邮件，邮件就是一组文本。你需要提取特征，比如：

- 邮件中是否包含单词“Viagra”；
- 字母 d 出现了多少次；
- 寄件人的域名是什么。

第一个问题的特征就是简单的是或否，可以被典型地编码为 1 或 0。第二个问题的特征是个数字。第三个问题的特征是从一个离散的选项集中做出的选择。

多数情况下，我们会从符合这三种特征的数据中提取特征。此外，特征的类型限制了我们所用模型的类型。

第 13 章中使用的朴素贝叶斯分类器适合“是或否”这样的二元特征，就像上面列出的第一种情况一样。

第 14 章和第 16 章将要提到的回归模型要求有数值型的特征（它可能会包括 0 或 1 这样的虚拟变量）。

第 17 章讲到的决策树，会涉及数值或属性数据。

尽管在垃圾邮件过滤器的例子中我们探索了创建特征的方法，但有时我们还需要设法移除特征。

比如，输入可能是包含几百个数的向量。根据具体情况，最好是取出一些维度，缩减到只剩少量重要的维度（见 10.5 节“降维”），只使用这些少数的特征，或者最好采用一些技术（比如正则化技术，见 15.8 节“正则化”）对应用过多特征的模型进行惩罚。

我们该如何选择特征呢？这需要经验和专业知识的结合。如果你收到了大量的邮件，可能会对某个特定的词比较敏感，这个词会成为垃圾邮件的好指标。同时，你也可能会觉得，字母 d 的个数不像是判断垃圾邮件的好指标。但通常来说，你需要尝试不同的特征，这也不失为一种乐趣。

11.7 延伸学习

- 继续读下去，后面几章讨论了不同种类的机器学习模型。
- Coursera 的机器学习课程（https://www.coursera.org/learn/machine-learning）是原创的 MOOC，是深入理解机器学习基础知识的好途径。加州理工学院的机器学习 MOOC（https://work.caltech.edu/telecourse.html）也是很好的资源。
- *The Elements of Statistical Learning* 是一本相当权威的教材，可以从网络上免费下载（http://statweb.stanford.edu/~tibs/ElemStatLearn/）。但是要注意，它是非常数学化的。

第 12 章

k 近邻法

惹怒你的邻居很容易，只需坦言你对他们的真实印象。

——皮特罗·雷提诺

假设你打算预测我会在下次大选给谁投票。如果你对我一无所知（但是你有数据），一个明智的方法是看看我的邻居们打算怎么投票。我住在西雅图市中心，我的邻居们总是会打算投票给民主党的候选人，这说明“民主党候选人”有可能是我的投票对象。

现在假设你对我的了解不仅限于家庭住址——可能你还知道我的年龄、收入，以及有几个孩子，等等。参照我的行为被这些维度影响（或刻画）的程度，观察那些在这些维度上最接近我的邻居似乎比观察我所有的邻居会得到更好的预测结果。这就是最近邻分类（nearset neighbors classification）方法背后的思想。

12.1 模型

最近邻法是最简单的预测模型之一，它没有多少数学上的假设，也不要求任何复杂的处理，它所要求的仅仅是：

- 某种距离的概念
- 一种彼此接近的点具有相似性质的假设

本书中讲到的大部分技术是把数据集看作一个整体，以便学习数据中的模式。相比之下，最近邻法却非常有意地忽略了大量信息，因为对每一个新的数据点进行预测只依赖于少量最接近它的点。

而且，最近邻法并不能帮助理解你所观察到的任意现象的驱动机制。比如，基于我邻居的投票行为来预测我的投票并不能告诉你我为什么要这样投票，而某些基于（比如说）我的收入和婚姻状况来预测我投票行为的模型很可能会揭示我投票的原因。

在一般情形中，我们有一些数据和对应的标签集。这些标签可能记作真或假，表示每个输入是否满足诸如“是否是垃圾邮件”“是否有毒”“是否值得观看”这样的条件；或者它们可能表示属性，就像“G、PG、PG-13、R、PG-17”这样的电影评分；或者它们可能是总统候选人的名字；或者它们可能是最受欢迎的编程语言的名称。

在我们的例子里，数据点是向量，这意味着可以用到第 4 章中的 `distance` 函数。

比如说我们已选定了数字 k 的值为 3 或 5，然后想要对某些新的数据点分类时，我们寻找 k 个已标记的最接近它的点，让这些点在新的输出上投票。

为此，我们需要一个函数来计算投票结果。一个可能的函数是：

```python
def raw_majority_vote(labels):
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner
```

但这里没有智能地处理并列的结果。比如，假设我们在对电影评分，且 5 个最接近的电影被评为 G、G、PG、PG 和 R，那么 G 有两票，PG 也有两票。在这种情形下，我们有以下几种选择。

- 随机选择其中一个获胜者。
- 根据距离加权投票并选择加权的获胜者。
- 减少 k 值直到找到唯一的获胜者。

我们将运用第三种方法：

```python
def majority_vote(labels):
    """assumes that labels are ordered from nearest to farthest"""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])

    if num_winners == 1:
        return winner                     # 唯一的获胜者,返回它的值
    else:
        return majority_vote(labels[:-1]) # 去掉最远元素,再次尝试
```

这种方法最终是管用的，因为在最坏的情况下，我们会一直减少 k 值，直到只剩一个标签，此时这个标签就是获胜者。

使用这个函数很容易创建一个分类器：

```
def knn_classify(k, labeled_points, new_point):
    """each labeled point should be a pair (point, label)"""

    # 把标记好的点按从最近到最远的顺序排序
    by_distance = sorted(labeled_points,
                         key=lambda (point, _): distance(point, new_point))

    # 寻找k个最近邻的标签
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # 然后让它们投票
    return majority_vote(k_nearest_labels)
```

让我们看看这是怎么起作用的。

12.2　案例：最喜欢的编程语言

DataSciencester 第一次用户调查的结果回来了，我们从中找到了一系列大城市用户偏爱的编程语言：

```
# 每一条记录都是([longitude, latitude], favorite_language)的形式

cities = [([-122.3 , 47.53], "Python"),  # 西雅图
          ([-96.85, 32.85], "Java"),     # 奥斯汀
          ([ -89.33, 43.13], "R"),       # 麦迪逊
          # ……还有很多记录
]
```

社区参与部门的副总想知道我们能不能用这些结果来预测那些我们没有调查到的地方最喜欢的编程语言是什么。

一如既往地，第一步最好是先根据数据作图（如图 12-1 所示）：

```
# 键是语言,值是成对数据(longitudes, latitudes)
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

# 我们希望每种语言都能有不同的记号和颜色
markers = { "Java" : "o", "Python" : "s", "R" : "^" }
colors  = { "Java" : "r", "Python" : "b", "R" : "g" }

for (longitude, latitude), language in cities:
    plots[language][0].append(longitude)
    plots[language][1].append(latitude)

# 对每种语言创建一个散点序列
for language, (x, y) in plots.iteritems():
    plt.scatter(x, y, color=colors[language], marker=markers[language],
                label=language, zorder=10)
```

```
plot_state_borders(plt)       # 假设我们有一个实现这一步的函数

plt.legend(loc=0)             # 让matplotlib选择一个位置
plt.axis([-130,-60,20,55])    # 设置轴

plt.title("最受欢迎的编程语言")
plt.show()
```

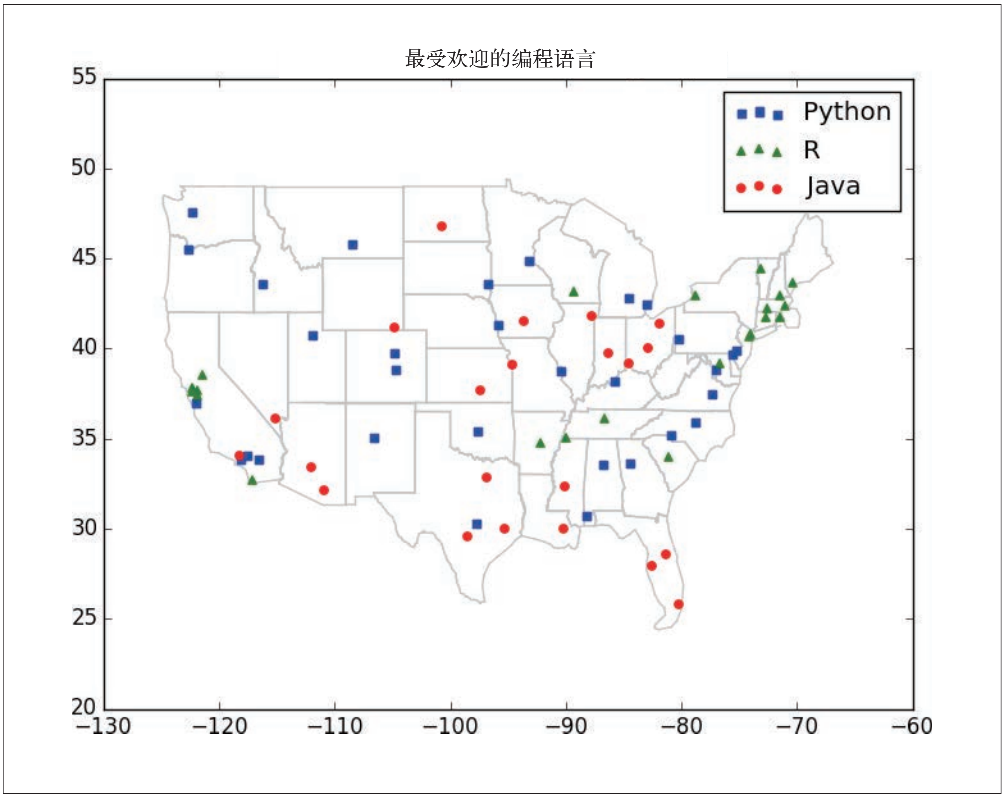


图 12-1：最受欢迎的编程语言

你可能注意到了对 `plot_state_borders()` 的调用，这是一个没有被精确定义的函数。本书的 GitHub 页面（https://github.com/joelgrus/data-science-from-scratch）上有这个函数的具体实现，你也可以把它当作一个练习题来自己尝试解决：

(1) 在网络上搜索各州边界线的经纬度等信息；

(2) 把你找到的任意经纬度数据转化为线段 [(long1, lat1), (long2, lat2)] 的列表；

(3) 使用 `plt.plot()` 画出这些线段。

既然互相邻近的地区看起来偏爱相同的语言，那么 *k* 近邻法作为一种预测模型看上去会是一种合理的选择。

首先，让我们看一下如果尝试利用邻居城市来预测每个城市偏爱的语言会得到什么结果：

```
# 试试多个不同的k值
for k in [1, 3, 5, 7]:
    num_correct = 0

    for city in cities:
        location, actual_language = city
        other_cities = [other_city
                        for other_city in cities
                        if other_city != city]

        predicted_language = knn_classify(k, other_cities, location)

        if predicted_language == actual_language:
            num_correct += 1

     print k, "neighbor[s]:", num_correct, "correct out of", len(cities)
```

看起来 3- 近邻的表现最好，59% 的时间都能给出正确结果：

```
1 neighbor[s]: 40 correct out of 75
3 neighbor[s]: 44 correct out of 75
5 neighbor[s]: 41 correct out of 75
7 neighbor[s]: 35 correct out of 75
```

现在可以看出在每个最近邻体系下会把某个区域分类到哪种语言。我们可以在全部的网格点上进行这种分类，然后参照处理城市分类的方法把预测结果画出来：

```
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

k = 1 # 或3,或5,或……

for longitude in range(-130, -60):
    for latitude in range(20, 55):
        predicted_language = knn_classify(k, cities, [longitude, latitude])
        plots[predicted_language][0].append(longitude)
        plots[predicted_language][1].append(latitude)
```

例如，图 12-2 显示了当我们只看最近的邻居（*k*=1）时会有什么结果。

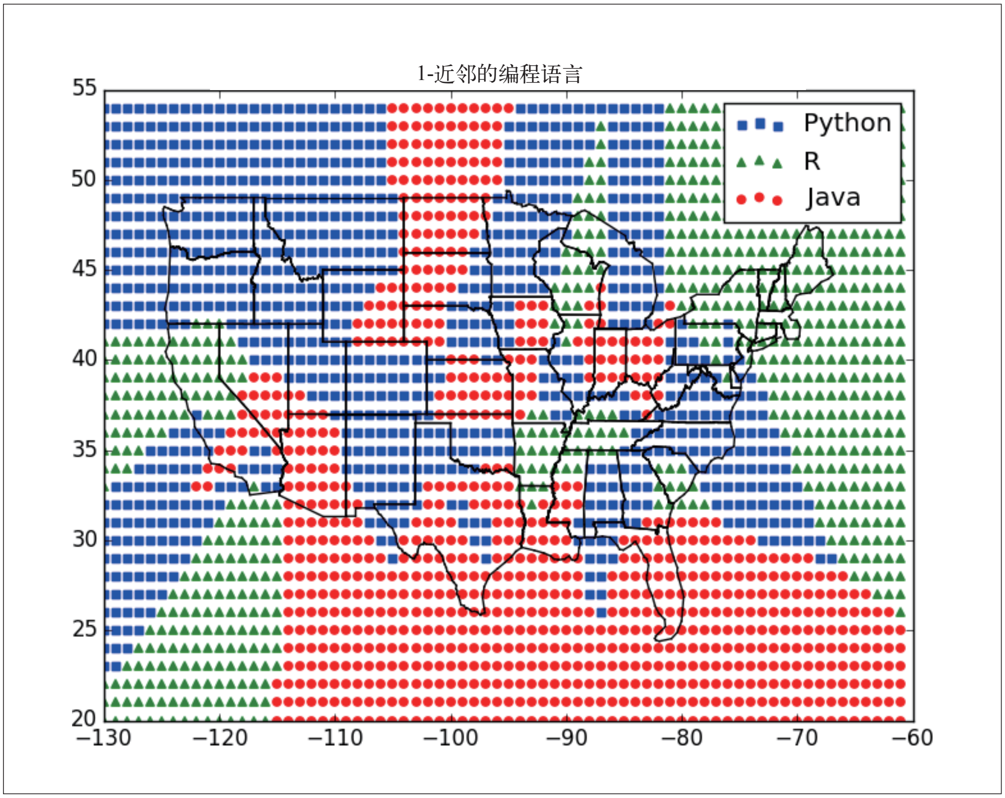


图 12-2：1- 近邻的编程语言

可以看到，从一种语言到另一种语言有许多骤变，它们之间的边界也较为锐化。当我们把邻居数增加到 3 时，能看到各种语言的区域变光滑了（图 12-3）。

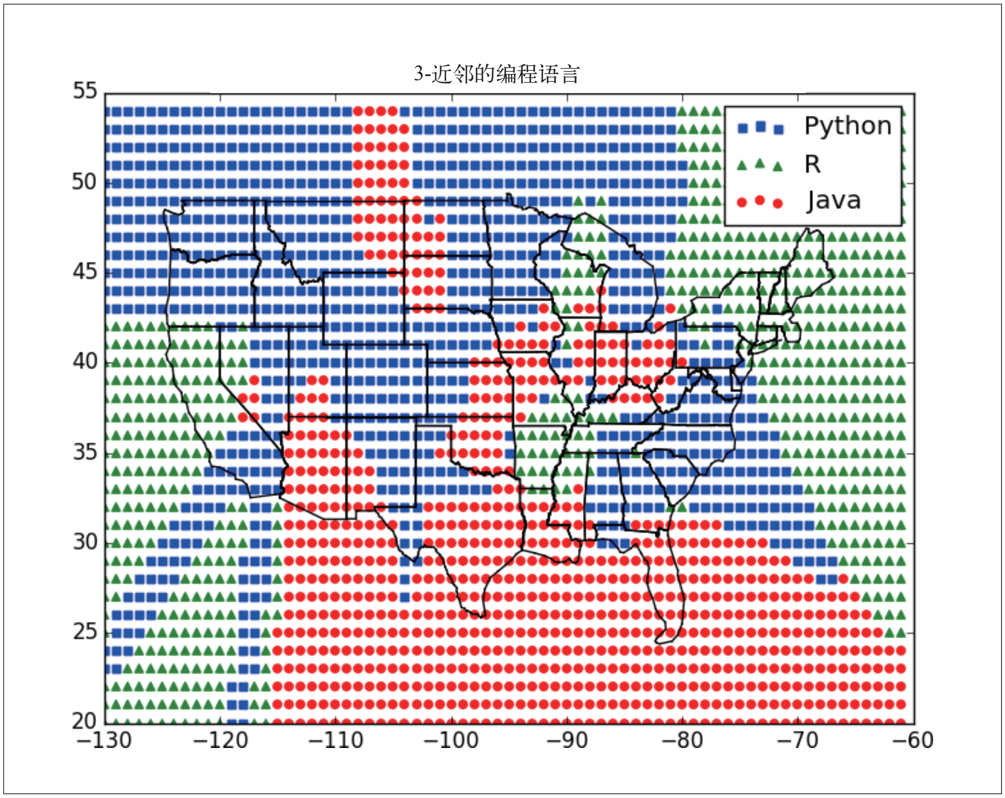


图 12-3：3- 近邻的编程语言

我们把邻居数增加到 5，边界变得更加光滑了（图 12-4）。

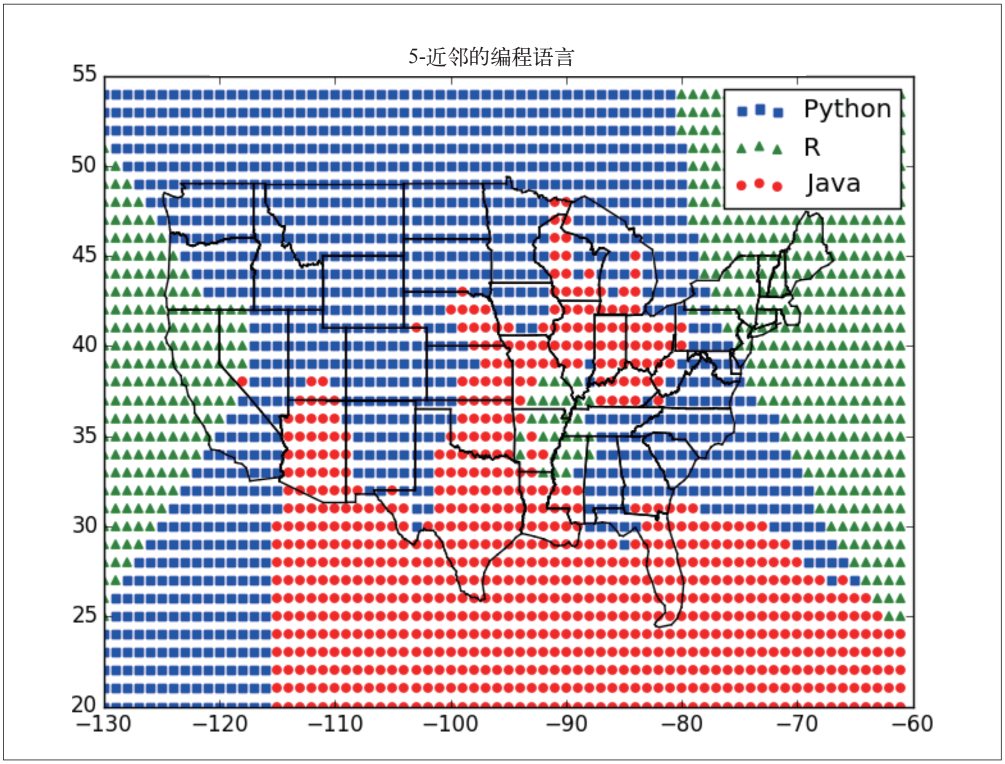


图 12-4：5- 近邻的编程语言

在这里，维度是大致可比较的，如果不是这样，你可能需要重新调整数据，就像我们在 10.4 节“数据调整”中所讲的那样。

12.3 维数灾难

在更高的维度上，*k* 近邻法会因为“维数灾难”而遇到麻烦，其根源在于高维空间过于巨大。高维空间内的点根本不会表现得彼此邻近。观察维数灾难的一种方法是在一个高维度的 d 维空间“单位立方体”上随机地生成数据点对，并计算它们之间的距离。

现在，生成随机点应该是老生常谈了：

```
def random_point(dim):
    return [random.random() for _ in range(dim)]
```

写一个函数生成距离也是如此：

```
def random_distances(dim, num_pairs):
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

对从 1 到 100 的每一个维度，我们会计算 10 000 个距离，并使用它们计算每个维度上点和点之间的平均距离和最小距离（图 12-5）：

```
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in dimensions:
  distances = random_distances(dim, 10000)   # 10 000个随机对
  avg_distances.append(mean(distances))      # 追踪平均值
  min_distances.append(min(distances))       # 追踪最小值
```

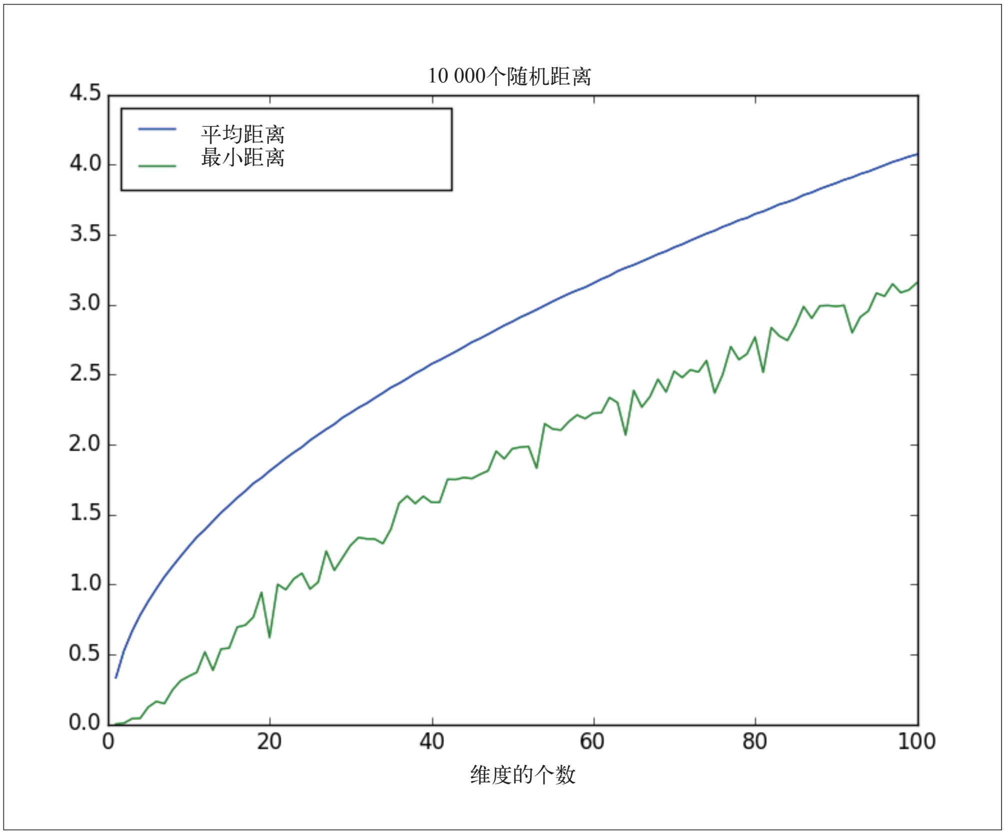


图 12-5：维数的灾难

随着维度数量的增加，点和点之间的平均距离也增加了。但更麻烦的是最近距离和平均距离之间的比例（图 12-6）：

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

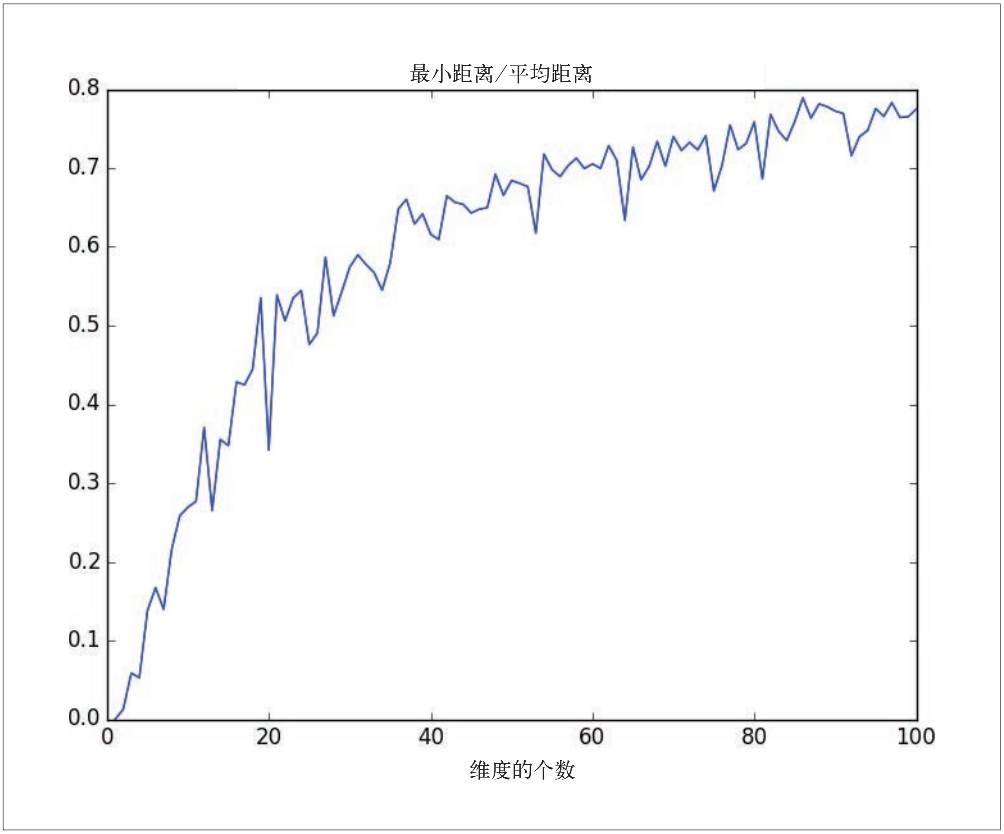


图 12-6：维数的另一个灾难

在低维数据集中，最邻近的点的距离看起来比点和点的平均距离要更小。但仅当两个点在每个维度上都邻近时，我们才可称这两个点是邻近的，而且每个增加的维度——即使仅仅是噪声——都有可能会让每个点更加远离其他的点。当有许多维度时，看上去最邻近的两个点的距离并不比点和点的平均距离小，这说明两个点邻近并不特别意味着什么（数据中有许多结构的行为使其看起来像是在更低的维度）。

思考这个问题的一个不同的方法涉及更高维空间的稀疏性。

如果从 0 到 1 之间随机取 50 个数，你可能会得到单位区间内的一个非常好的样本（图 12-7）。

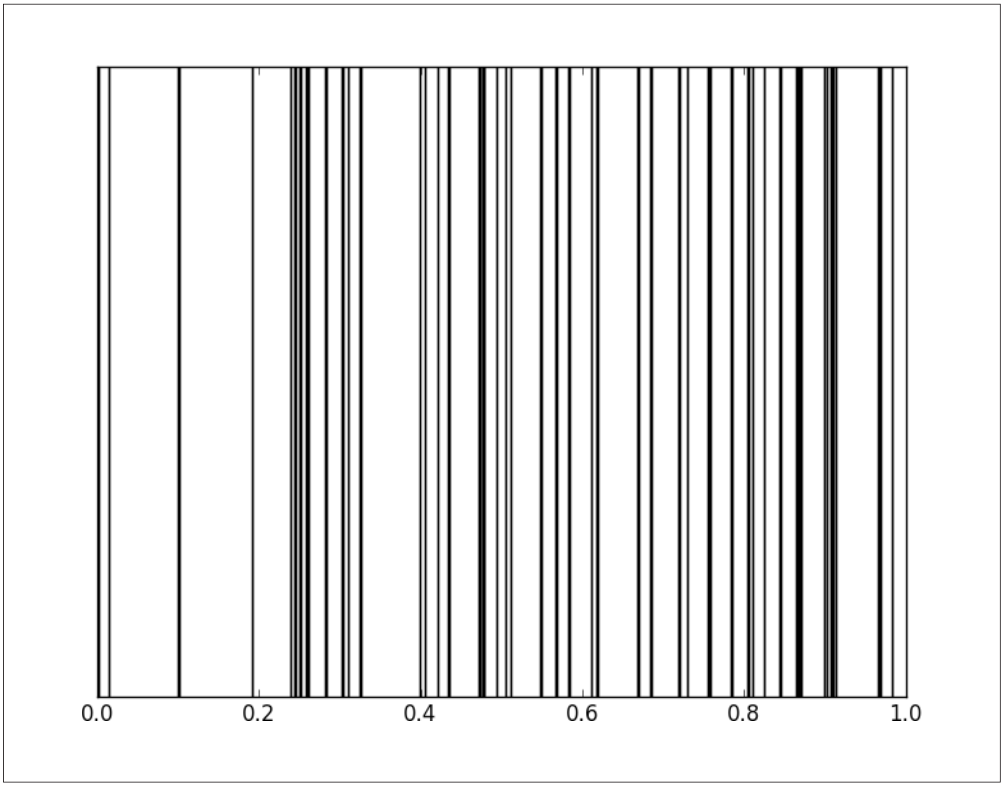


图 12-7：一维内的 50 个随机点

如果在单位正方形内随机取 50 个点，得到的规模会更小（图 12-8）。

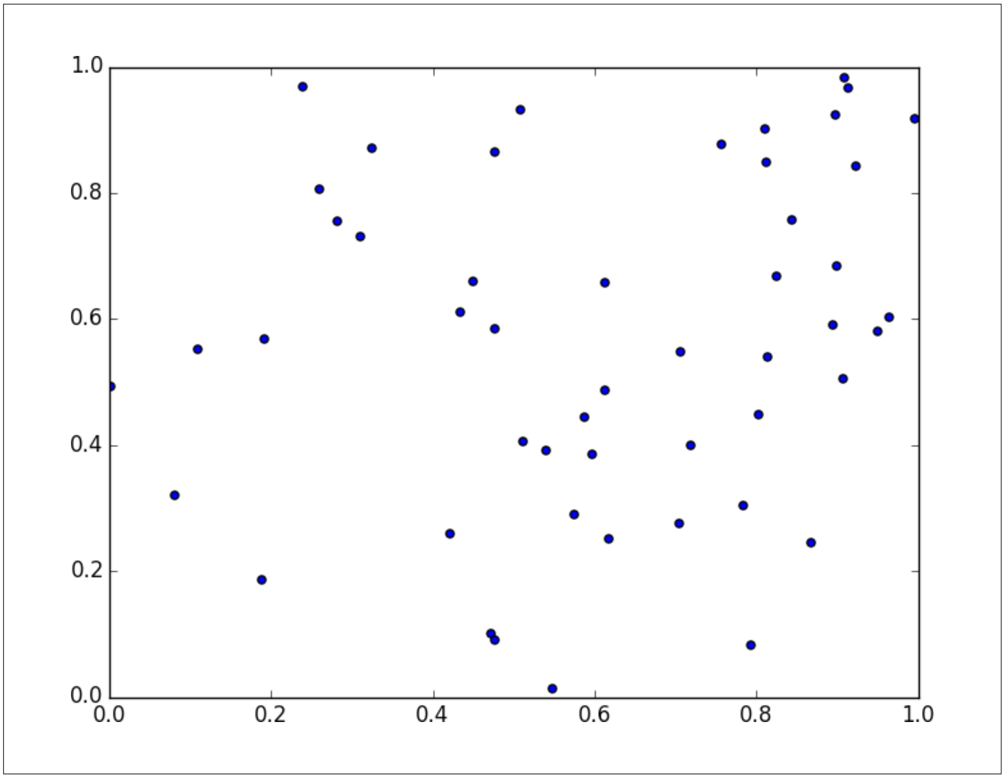


图 12-8：二维内的 50 个随机点

在三个维度中的随机样本会变得更稀疏（图 12-9）。

`matplotlib` 不能很好地画出 4 维的图形，所以我们只能给出上面的几种情形，即便如此，你也已经能够看到某些点的附近因为没有邻近的点而存在大片的空白空间。在更高的维度上——除非你能以指数规模得到更多的数——大片空白空间代表的是远离你想用在预测中的所有的点的区域。

因此，如果你打算在高维中使用最近邻法，不妨先做一些降维工作。

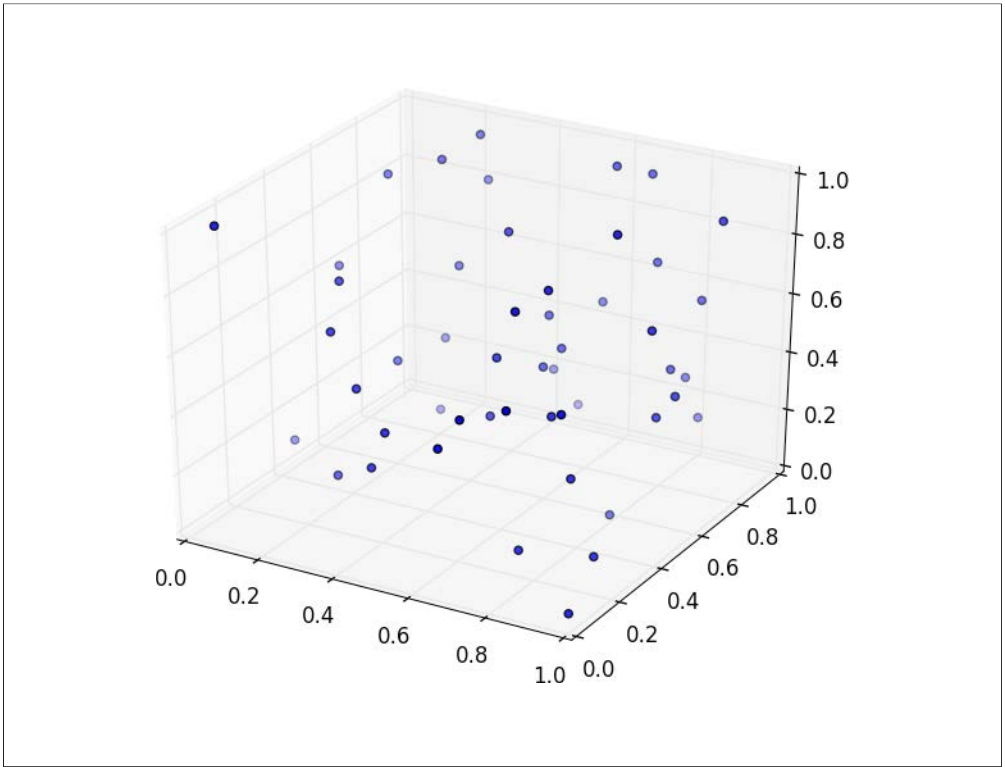


图 12-9：三维内的 50 个随机点

12.4　延伸学习

scikit-learn 里有许多最近邻模型（http://scikit-learn.org/stable/modules/neighbors.html）。

第13章

朴素贝叶斯算法

> 心灵朴素多为巧，头脑朴素却成拙。
>
> ——阿纳托尔·法郎士

如果人们不形成关系网的话，社交网络就不会有太大的用途。因此，DataSciencester 提供了一个非常流行的功能，允许会员之间相互发送邮件。虽然大部分会员都是发邮件嘘寒问暖的良民，但是，总少不了有几个坏家伙，老给其他会员发送垃圾邮件，如致富经、药品广告以及以收费为目的的数据科学家资格认证项目等。因此，用户们开始投诉。不久，邮件服务部的副总就把你叫过去，让你利用数据科学来过滤这些垃圾邮件。

13.1　一个简易的垃圾邮件过滤器

想象有一个“全集”，其中存放了从所有可能的邮件中随机选择的邮件。我们令 S 表示事件“这是一封垃圾邮件”，令 V 表示事件“该邮件含有单词 viagra”。在已知邮件中含有单词 viagra 的情况下，该邮件是垃圾邮件的概率可以通过贝叶斯定理求出：

$$P(S|V)=[P(V|S)P(S)]/[P(V|S)P(S)+P(V|\neg S)P(\neg S)]$$

上式中，分子表示某邮件为垃圾邮件并且其中包含单词 viagra 的概率，而分母表示邮件中出现单词 viagra 的概率。因此，你可以认为，上面的公式实际上是在计算兜售伟哥的垃圾邮件所占的比例。

如果我们已经收集了大量垃圾邮件和非垃圾邮件，那么就可以轻松计算 $P(V\mid S)$ 和 $P(V\mid\neg S)$。如果我们进一步假定任何邮件是垃圾邮件或非垃圾邮件的可能性是等同的

（即 $P(S)=P(\neg S)=0.5$），那么：

$$P(S|V)=P(V|S)/[P(V|S)+P(V|\neg S)]$$

举例来说，如果 50% 的垃圾邮件都含有单词 viagra，而只有 1% 的非垃圾邮件含有该单词，那么任何一封含有单词 viagra 的电子邮件为垃圾邮件的概率是：

$$0.5/(0.5+0.01)=98\%$$

13.2 一个复杂的垃圾邮件过滤器

假设我们已建立了一个词汇表，其中含有许多单词：$w_1, \cdots, w_n$。站在概率论的角度，我们用 X_i 表示事件“一封含有单词 w_i 的邮件”。此外，我们还假设已经求出了 $P(X_i|S)$ 和 $P(X_i|\neg S)$，前者表示垃圾邮件中出现第 i 个单词的概率，后者表示非垃圾邮件中出现第 i 个单词的概率。

朴素贝叶斯算法的一个（大的）假设是，给定邮件是或不是垃圾邮件的条件下，其中的每个单词存在与否与其他单词毫不相干。直观地讲，就是知道某封垃圾邮件是否含有单词 viagra 无法帮助我们判断该垃圾邮件是否含有单词 rolex。如果用数学公式表示的话，就是：

$$P(X_1=x_1,\cdots,X_n=x_n|S)=P(X_1=x_1|S)\times\cdots\times P(X_n=x_n|S)$$

这是一个非常极端的假设（这也部分解释了为何该算法名中含有“朴素”一词）。假设我们的词汇表仅含有单词 viagra 和 rolex，并且一半的垃圾邮件是推销“廉价伟哥”的，另一半是推销“劳力士正品”的，这样的话，我们可以通过朴素贝叶斯算法计算垃圾邮件中同时出现 viagra 和 rolex 这两个单词的概率：

$$P(X_1=1,X_2=1|S)=P(X_1=1|S)P(X_2=1|S)=0.5\times0.5=0.25$$

之所以得到这样的结果，是因为我们的假设已经把 viagra 和 rolex 绝不会同时出现的经验给扔掉了。尽管这个假设与事实并不相符，但是这个模型的表现通常都很好，所以现实中经常用它过滤垃圾邮件。

前面我们曾经利用贝叶斯定理过滤只涉及 viagra 的垃圾邮件，下面我们再次用它来推断一个邮件是垃圾邮件的概率，具体公式如下所示：

$$P(S|X=x)=P(X=x|S)/[P(X=x|S)+P(X=x|\neg S)]$$

朴素贝叶斯假设使我们能够轻松求出公式右边的每个概率：只要将词汇表中各个单词的概率相乘即可。

在实践中，为了避免所谓的下溢（underflow）问题，你通常希望尽量避免出现大量概率相乘的情况，因为计算机不擅长处理非常接近于零的浮点数。根据代数知识我们知道，

$\log(ab)=\log a+\log b$ 且 $\exp(\log x)=x$，因此我们一般使用对浮点数更加友好的等效方法来计算 $p_1*\cdots*p_n$，具体公式如下所示：

$$\exp(\log(p_1)+\cdots+\log(p_n))$$

现在，唯一的挑战就是估计 $P(X_i|S)$ 和 $P(X_i|\neg S)$ 了，即估计垃圾邮件（或非垃圾邮件）中包含单词 w_i 的概率。如果我们掌握了相当数量的“训练”邮件，即标记为垃圾或非垃圾的邮件，那么很明显，这时计算 $P(X_i|S)$ 就简化为求包含单词 w_i 的垃圾邮件所占的比例了。

但是，这会引起一个大麻烦。假如词汇表中的单词 data 仅出现在训练集的非垃圾邮件中，那么 $P($“data”$|S)=0$。也就是说，对于任何含有单词 data 的邮件，我们的朴素贝叶斯分类器总是认为它是垃圾邮件的概率为 0，即使是像含有“data on cheap viagra and authentic rolex watches”（关于廉价伟哥和劳力士正品的数据）这样的邮件也是如此。为了避免这种问题，我们通常要使用某种平滑技术。

准确地说，我们引入一个*伪记数*（pseudo count），记为 k，并通过下面的公式来计算在一封垃圾邮件中出现第 i 个单词的概率：

$$P(X_i|S) = (k + \text{含有 } w_i \text{ 的垃圾邮件的数量}) / (2k + \text{垃圾邮件数量})$$

$P(X_i|\neg S)$ 的计算方法与此类似。亦即，当计算第 i 个单词出现在垃圾邮件中的概率时，我们假定还看到：额外 k 封垃圾邮件包含该单词，额外 k 封垃圾邮件不包含该单词。

例如，如果 data 这个单词在 98 封垃圾邮件中出现了 0 次，并且 k 取值为 1，我们算出 $P($“data”$|S)$ 为 $1/100 = 0.01$，这样一来，我们的分类器就能给那些含有单词 data 的邮件为垃圾邮件的概率赋予非 0 值了。

13.3 算法的实现

到目前为止，我们已经学习了构建垃圾邮件分类器所需的各方面的知识。下面，我们首先建立一个简单的函数，来将邮件解析为不同的单词。首先要把各个邮件文本转换为小写形式，然后使用 `re.findall()` 提取由字母、数字和撇号组成的“单词”，最后使用 `set()` 函数获得不同的单词：

```
def tokenize(message):
    message = message.lower()                          # 转换为小写
    all_words = re.findall("[a-z0-9']+", message)      # 提取单词
    return set(all_words)                              # 移除副本
```

我们的第二个函数用来计算单词出现在已做标记的邮件训练集中的次数。该函数将返回一个字典，其键为单词，其值为列表，该列表包含两个元素 `[spam_count, non_spam_count]`，分别表示该单词出现在垃圾邮件和非垃圾邮件中的次数。

```
def count_words(training_set):
    """training set consists of pairs (message, is_spam)"""
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in tokenize(message):
            counts[word][0 if is_spam else 1] += 1
    return counts
```

接下来，我们利用前面讲过的平滑技术将这些计数转换为估计概率。函数将返回一个列表，列表元素包含三方面的内容，分别是各个单词、该单词出现在垃圾邮件中的概率以及该单词出现在非垃圾邮件中的概率：

```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
    """turn the word_counts into a list of triplets
    w, p(w | spam) and p(w | ~spam)"""
    return [(w,
             (spam + k) / (total_spams + 2 * k),
             (non_spam + k) / (total_non_spams + 2 * k))
             for w, (spam, non_spam) in counts.iteritems()]
```

最后要做的事情是利用这些单词的概率（以及朴素贝叶斯假设）给邮件赋予概率：

```
def spam_probability(word_probs, message):
    message_words = tokenize(message)
    log_prob_if_spam = log_prob_if_not_spam = 0.0

    # 迭代词汇表中的每一个单词
    for word, prob_if_spam, prob_if_not_spam in word_probs:

        # 如果*word*出现在了邮件中
        # 则增加看到它的对数概率
        if word in message_words:
            log_prob_if_spam += math.log(prob_if_spam)
            log_prob_if_not_spam += math.log(prob_if_not_spam)

        # 如果*word*没有出现在邮件中
        # 则增加看不到它的对数概率
        # 也就是log(1 - 看到它的概率)
        else:
            log_prob_if_spam += math.log(1.0 - prob_if_spam)
            log_prob_if_not_spam += math.log(1.0 - prob_if_not_spam)

    prob_if_spam = math.exp(log_prob_if_spam)
    prob_if_not_spam = math.exp(log_prob_if_not_spam)
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)
```

将上面的代码结合起来，就得到了我们的朴素贝叶斯分类器：

```
class NaiveBayesClassifier:

    def __init__(self, k=0.5):
        self.k = k
        self.word_probs = []
```

```python
    def train(self, training_set):

        # 对垃圾邮件和非垃圾邮件计数
        num_spams = len([is_spam
                         for message, is_spam in training_set
                         if is_spam])
        num_non_spams = len(training_set) - num_spams

        # 通过"pipeline"运行训练数据
        word_counts = count_words(training_set)
        self.word_probs = word_probabilities(word_counts,
                                             num_spams,
                                             num_non_spams,
                                             self.k)

    def classify(self, message):
        return spam_probability(self.word_probs, message)
```

13.4 测试模型

SpamAssassin 垃圾邮件公共语料库（https://spamassassin.apache.org/publiccorpus/）是一个非常不错（尽管有点老）的数据集。我们将考察其中前缀为 20021010 的文件。在 Windows 系统上，你可能需要用到类似 7-Zip（http://www.7-zip.org/）的压缩软件来解压和提取文件。

提取数据之后（例如提取到 C:\spam 目录下面），你会看到 3 个文件夹：spam、easy_ham 和 hard_ham。每个文件夹中都存放了许多电子邮件，每封邮件都单独存放于一个文件之中。为简单起见，我们只检测每封邮件的主题行。

那么，我们应如何识别主题呢？通过观察可以发现，这些文件似乎都是以“Subject：”开头的。因此，我们可以利用下面的代码来识别主题内容：

```python
import glob, re

# 把路径修改为你存放文件的那个
path = r"C:\spam\*\*"

data = []

# glob.glob会返回每一个与通配路径所匹配的文件名
for fn in glob.glob(path):
    is_spam = "ham" not in fn

    with open(fn,'r') as file:
        for line in file:
            if line.startswith("Subject:"):
                # 移除开头的"Subject: ",保留其余内容
                subject = re.sub(r"^Subject: ", "", line).strip()
                data.append((subject, is_spam))
```

好了，现在我们把数据分为训练数据和测试数据，然后开始建立分类器：

```
random.seed(0)      # 这样你能得到与我相同的答案
train_data, test_data = split_data(data, 0.75)

classifier = NaiveBayesClassifier()
classifier.train(train_data)
```

然后，我们可以检查一下模型的效果如何：

```
# 三个元素（主题,确实是垃圾邮件,预测为垃圾邮件的概率）
classified = [(subject, is_spam, classifier.classify(subject))
              for subject, is_spam in test_data]

# 假设spam_probability > 0.5对应的是预测为垃圾邮件
# 对(actual is_spam, predicted is_spam)的组合计数
counts = Counter((is_spam, spam_probability > 0.5)
                 for _, is_spam, spam_probability in classified)
```

结果显示，真阳性（即垃圾邮件被分类为 spam）有 101 例，假阳性（即正常邮件被分类为 spam）有 33 例，真阴性（即正常邮件被分类为 ham）有 704 例，以及假阴性（即垃圾邮件被分类为 ham）有 38 例。也就是说，算法的查准率是 101 / (101 + 33) = 75%，查全率是 101 / (101 + 38) = 73%，对于如此简单的一个模型来说，这样的结果已经不错了。

由此也引出了一个有趣的问题，到底哪些邮件最容易被错误分类呢？请看下面的代码：

```
# 根据spam_probability从最小到最大排序
classified.sort(key=lambda row: row[2])

# 非垃圾邮件被预测为垃圾邮件的最高概率
spammiest_hams = filter(lambda row: not row[1], classified)[-5:]

# 垃圾邮件被预测为垃圾邮件的最低概率
hammiest_spams = filter(lambda row: row[1], classified)[:5]
```

这两种最容易被误判为垃圾邮件的正常邮件都含有单词 needed（它在垃圾邮件中出现的概率要高 77 倍）、insurance（它在垃圾邮件中出现的概率要高 30 倍）和 important（它在垃圾邮件中出现的概率要高 10 倍）。

最容易误判为正常邮件的垃圾邮件的标题都太短（“Re: girls”），以至于难以判断；排行第二的容易误判为正常邮件的垃圾邮件是信用卡邀约邮件，因为相关的词大多尚未被收录到训练集中。

同样，我们也可以看出现哪些词最容易被误判为垃圾邮件，具体代码如下所示：

```
def p_spam_given_word(word_prob):
    """uses bayes's theorem to compute p(spam | message contains word)"""

    # word_prob是由word_probabilities生成的三元素中的一个
```

```
        word, prob_if_spam, prob_if_not_spam = word_prob
        return prob_if_spam / (prob_if_spam + prob_if_not_spam)

    words = sorted(classifier.word_probs, key=p_spam_given_word)

    spammiest_words = words[-5:]
    hammiest_words = words[:5]
```

最容易被判定为垃圾邮件的单词包括 money、systemworks、rates、sale 以及 year，这些似乎都与忽悠人买东西有关。而最容易被判定为正常邮件的单词有 spambayes、users、razor、zzzzteana 和 sadev，其中大部分好像都与阻止垃圾邮件相关，这比较奇怪。

那么，我们该如何改进性能呢？一个显而易见的方法是设法获取更多的训练数据。此外，还有许多可以改善模型本身的方法。大家不妨尝试以下具体的方法。

- 考察邮件内容，而不是仅仅考察邮件主题。你还必须仔细考虑邮件开头的处理方式。
- 我们的分类器考虑了训练集中包含的所有单词，即使该单词仅仅出现过一次。修改分类器，让它接受一个可选阈值 `min_count`，并且如果某个单词在训练集中出现次数少于阈值则不予考虑。
- 标记赋予器缺乏相似词（例如 cheap 和 cheapest）的概念。修改分类器，使其接受一个可选的词干分析器（`stemmer`）函数来找出单词对应的同类词。下面我们以一个非常简单的词干分析器函数为例进行介绍：

  ```
  def drop_final_s(word):
      return re.sub("s$", "", word)
  ```

  自己创建一个非常好用的词干分析器函数非常困难，所以人们通常使用现成的波特词干器（Porter Stemmer，http://tartarus.org/martin/PorterStemmer/）。

- 虽然我们的特征都是采取了“含有单词 w_i 的邮件”的形式，但这不代表必须采取这种形式。在我们的代码实现中，也可以添加额外的特征，比如“含有一个数字的邮件”。为此，可以创建类似 contains:number 这样的伪标记，然后修改标记赋予器（`tokenizer`），让它在适当的时候放出这些伪标记。

13.5 延伸学习

- Paul Graham 的“A Plan for Spam”（http://www.paulgraham.com/spam.html） 和“Better Bayesian Filtering”（http://www.paulgraham.com/better.html）这两篇文章对如何打造垃圾邮件过滤器提供了更加有趣而深入的介绍。
- `scikit-learn` 库（http://scikit-learn.org/stable/modules/naive_bayes.html）提供了一个名为 `BernoulliNB` 的模型，也实现了与本章介绍的相同的朴素贝叶斯算法以及基于该算法的其他变种。

第 14 章

简单线性回归

> 艺术就像道德，需要在某个地方画线。
>
> ——吉尔伯特 · 切斯特顿

在第 5 章中，我们曾经使用相关函数 `correlation` 来衡量两个变量之间的线性关系的强度。对于大多数应用来说，仅仅知道存在这样的线性关系是远远不够的。如果我们希望了解这种关系的性质，可以使用简单线性回归。

14.1　模型

别忘了，我们正在探讨的是 DataSciencester 用户的朋友数量与其每天花在该网站上的时间之间的关系。我们假设，你已经说服自己结交的朋友越多会导致人们花在网站上的时间越长。

这时，参与部的副总要你建立一个模型来描述这种关系。既然你发现有很强的线性关系，那么自然就要从线性模型开始着手了。准确地说，假设有常数 α（alpha）和 β（beta），使得：

$$y_i=\beta x_i+\alpha+\varepsilon_i$$

其中，y_i 是用户 i 每天花在网站上的分钟数，x_i 是用户 i 已有的朋友数，而 ε_i 是误差项，用来表示这个简单模型没有考虑到的其他因素，当然，误差项越小越好。

只要我们求出 `alpha` 和 `beta`，就能轻松通过下列公式来进行预测了：

```
def predict(alpha, beta, x_i):
    return beta * x_i + alpha
```

那么，该如何选择 alpha 和 beta 呢？实际上，只要任意选定的 alpha 和 beta 值，对于每个输入 x_i，都能得到一个预测的输出值。由于知道实际输出值 y_i，因此可以计算它们的误差：

```
def error(alpha, beta, x_i, y_i):
    """the error from predicting beta * x_i + alpha
    when the actual value is y_i"""
    return y_i - predict(alpha, beta, x_i)
```

实际上，我们真正想知道的是整个数据集的总体误差情况。不过，我们不能简单地将各个误差加起来，这是因为，如果 x_1 预测得太高，而 x_2 预测得太低，那么它们的误差加在一起就会相互抵消了。

因此，我们要对误差的平方求和：

```
def sum_of_squared_errors(alpha, beta, x, y):
    return sum(error(alpha, beta, x_i, y_i) ** 2
               for x_i, y_i in zip(x, y))
```

我们可以通过最小二乘法来选择 alpha 和 beta，以使 sum_of_squared_errors 尽可能小。

利用微积分（或单调乏味的代数），我们就可以求出令误差最小化的 alpha 和 beta 了，具体代码如下所示：

```
def least_squares_fit(x, y):
    """given training values for x and y,
    find the least-squares values of alpha and beta"""
    beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta
```

不要忙着进行严格的数学推导，让我们先想想为什么这可能是一个合理的解决方案。实际上，选定 alpha 后，只要给出自变量 x 的平均值，我们就能预测因变量 y 的平均值。

选定了 beta，就意味着输入值每增加 standard_deviation(x)，预测值就会增加 correlation(x, y) * standard_deviation(y)。就本例来说，如果 x 和 y 完全相关，则 x 每增加一个标准偏差，预测值就会增加 y 的一个标准偏差。当它们完全负相关的时候，预测值会随着 x 的增加而减小。当它们的相关性为 0 时，beta 为 0，这意味着 x 的变化根本不会对预测值产生影响。

下面我们使用第 5 章中的异常值数据来计算这两个值：

```
alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)
```

计算结果是，alpha = 22.95，beta = 0.903。因此，根据我们的模型来看，具有 n 个好友的

用户每天会在这个网站上花 `22.95 + n * 0.903` 分钟。同时，对于在 DataSciencester 上面没有朋友的用户来说，他们每天仍然会花 23 分钟泡在这个网站上。此外，用户每增加一个朋友，每天花费在这个网站上的时间就会多出一分钟左右。在图 14-1 中，我们绘制了该模型的预测线，从中可以看出模型的预测与观测数据的拟合效果。

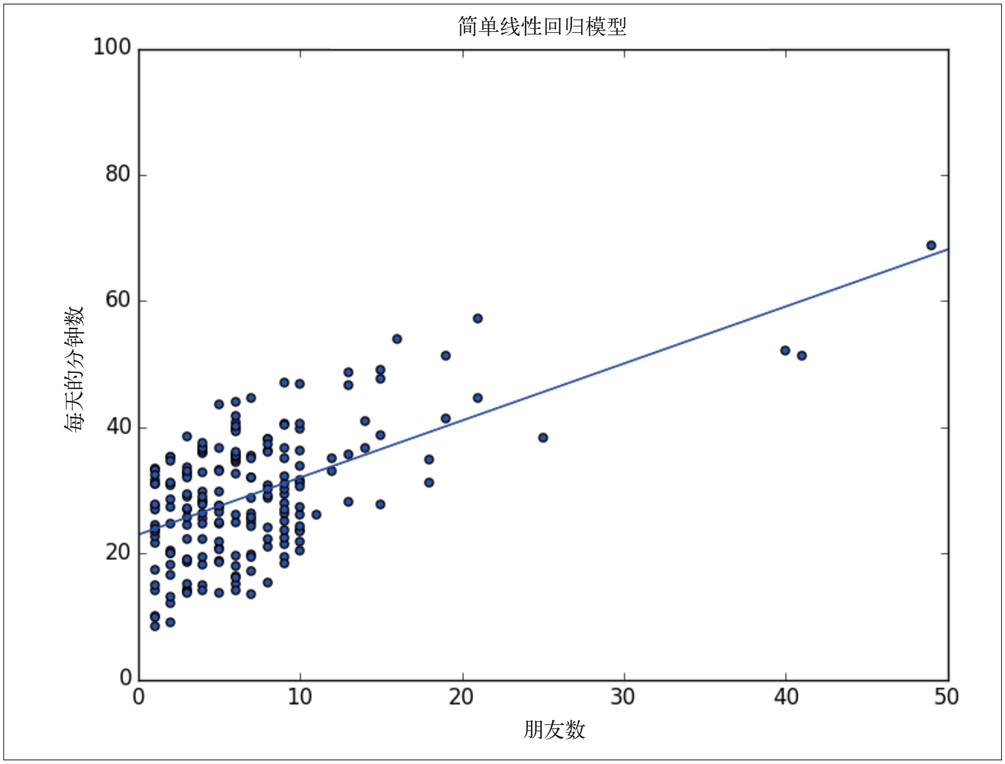


图 14-1：简单线性模型

当然，仅仅依靠目测是不够的，我们需要一个更好的指标来评估模型对数据的拟合效果。一个常见的指标是*决定系数*（coefficient of determination）或 R 平方，用来表示纳入模型的自变量引起的变动占总变动的百分比：

```
def total_sum_of_squares(y):
    """the total squared variation of y_i's from their mean"""
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha, beta, x, y):
    """the fraction of variation in y captured by the model, which equals
    1 - the fraction of variation in y not captured by the model"""

    return 1.0 - (sum_of_squared_errors(alpha, beta, x, y) /
                  total_sum_of_squares(y))

r_squared(alpha, beta, num_friends_good, daily_minutes_good)      # 0.329
```

现在，我们已经选取了令预测误差平方和最小化的 alpha 和 beta。我们选择的是一个“总是预测 mean(y)”的线性模型（即 alpha = mean(y) 且 beta = 0），模型误差平方和正好等于其平方总和。这就意味着拟合优度（R 平方）的值为 0，表明模型（显然，在这种情况下）几乎只能预测平均值。

很明显，最小二乘模型最差的时候，就是误差的平方和最大为平方总和的时候，也是 R 平方最小为 0 的时候。同时，因为误差的平方和至少为 0，所以 R 平方至多为 1。

R 平方的值越大，说明模型对数据的拟合度越高。在这里，R 平方的值为 0.329，说明模型对这些数据的拟合度不是很高，显然还有没考虑到的其他因素在起作用。

14.2 利用梯度下降法

如果我们记 theta = [alpha, beta]，那么也可以通过梯度下降法来求参数：

```
def squared_error(x_i, y_i, theta):
    alpha, beta = theta
    return error(alpha, beta, x_i, y_i) ** 2

def squared_error_gradient(x_i, y_i, theta):
    alpha, beta = theta
    return [-2 * error(alpha, beta, x_i, y_i),       # alpha偏导数
            -2 * error(alpha, beta, x_i, y_i) * x_i] # beta偏导数

# 选择一个随机值作为开始
random.seed(0)
theta = [random.random(), random.random()]
alpha, beta = minimize_stochastic(squared_error,
                                  squared_error_gradient,
                                  num_friends_good,
                                  daily_minutes_good,
                                  theta,
                                  0.0001)
print alpha, beta
```

使用相同的数据，我们得到 alpha = 22.93，beta = 0.905，这与精确的答案非常接近。

14.3 最大似然估计

我们为什么会选择最小二乘法呢？其中一个原因就是最大似然估计（maximum likelihood estimation）。假设我们的数据样本 $v_1,\cdots,v_n$ 服从由未知参数 θ 确定的概率分布：

$$p(v_1,\cdots,v_n|\theta)$$

虽然我们不知道 θ，但是可以回过头来通过给定样本与 θ 的相似度来考量这个参数：

$$L(\theta|v_1,\cdots,v_n)$$

按照这种方法，θ 最可能的值就是最大化这个似然函数的值，即能够以最高概率产生观测数据的值。在具有概率分布函数而非概率密度函数的连续分布的情况下，我们也可以做同样的事情。

再回到回归这个话题。对于简单回归模型来说，通常假设回归误差是呈正态分布的，其均值为 0，并且已知标准偏差 σ。 如果是这样的话，那么就可以通过下面的似然函数来描述 α 和 β 产生 (`x_i`，`y_i`) 的可能性大小了：

$$L(\alpha,\beta \mid x_i, y_i, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp(-(y_i - \alpha - \beta x_i)^2 / 2\sigma^2)$$

由于待估计的参数产生整个数据集的可能性为产生各个数据的可能性之积，因此令误差的平方和最小的 `alpha` 和 `beta` 最有可能是我们所求的。换句话说，在这种情况下（包括这些假设），最小化误差的平方和等价于最大化产生观测数据的可能性。

14.4 延伸学习

请继续往下阅读第 15 章介绍的多重回归分析！

第15章

多重回归分析

我不会盯着一个问题看，并往其中添加无用的变量。

——比尔 · 帕塞尔斯

虽然副总对你的预测模型很满意，但是他认为你还可以做得更好。为此，你收集了额外的数据：对于每一个用户，你不仅了解他每天工作多少小时，同时还调查了他是否拥有博士学位。你希望通过这些补充资料来改进模型。

因此，你提出了一个带有更多自变量的线性模型：

$$\text{minutes} = \alpha + \beta_1\text{friends} + \beta_2\text{work hours} + \beta_3\text{phd} + \varepsilon$$

显然，用户是否拥有博士学位并非一个数值问题，但如同第 11 章所提到的，我们可以引入一个虚拟变量，当这个变量等于 1 的时候，表示用户拥有博士学位，反之则表示没有博士学位，这样就能像其他变量一样将其视为一个数值了。

15.1 模型

回想一下，我们在第 14 章中所拟合的模型形式如下所示：

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

现在，如果每个输入 x_i 不再是单个数字，而是一个由 k 个数字 $x_i1, \cdots, x_ik$ 组成的向量，那么我们的多重回归模型则应该为：

$$y_i = \alpha + \beta_1 x_{i1} + \cdots + \beta_k x_{ik} + \varepsilon_i$$

对于多重回归分析来说，参数向量通常称为 β。我们希望这个向量也包括一个常数项，为此，只要向我们的数据中添加一列即可：

```
beta = [alpha, beta_1, ..., beta_k]
```

同时：

```
x_i = [1, x_i1, ..., x_ik]
```

那么，我们的模型可以用下列函数实现：

```
def predict(x_i, beta):
    """assumes that the first element of each x_i is 1"""
    return dot(x_i, beta)
```

就本例而言，自变量 x 是一个向量型列表，每个列表元素如下所示：

```
[1,    # 常数项
 49,   # 朋友数
 4,    # 每日工作时长
 0]    # 没有博士学位
```

15.2 最小二乘模型的进一步假设

对于我们的模型（以及我们的解决方案）来说，需要添加另外两个假设，才能够言之有理。

第一个假设是 x 的各个列是线性无关的，即任何一列绝对不会是其他列的加权和。如果这个假设不成立，则无法估计 beta。为了了解极端的情形，我们可以想象数据中有一个额外的字段 num_acquaintances，并且对于每一个用户来说它都等于 num_friends。

那么，对于任何 beta，如果 num_friends 的系数增加了某个数值，而 num_acquaintances 的系数同时减小相同数值的话，那么模型的预测就会保持不变。也就是说，我们根本就没有办法确定 num_friends 的系数。（通常来说，对于这个假设的违背情况一般不会这么明显。）

第二个重要的假设是 x 的各列与误差 ε 无关。如果这个假设不成立，对于 beta 的估计就会出现系统性的错误。

比如，在第 14 章中，我们建立的模型的预测结果为，用户每增加一个朋友，每天花在网站上的时间就会多出 0.90 分钟。

想象一下，还有下列情况。

- 工作时间越长的人在网站上花的时间越少。
- 朋友更多的人倾向于工作更长时间。

也就是说，假设“实际的”模型为：

$$\text{minutes} = \alpha + \beta_1\text{friends} + \beta_2\text{work hours} + \varepsilon$$

并且工作时间和朋友数量是正相关的。那样的话，当我们最小化单变量模型的误差时：

$$\text{minutes} = \alpha + \beta_1\text{friends} + \varepsilon$$

我们会低估 β_1。

考虑一下，如果这个单变量模型已知 β_1 的“实际”值，那么这时再用它来预测将会如何。（亦即，这个值来自令误差最小化的“实际”模型。）这时候，对工作时间比较长的用户来说，产生的预测值往往太小；对工作时间比较短的用户来说，产生的预测值往往又过大，这是因为 $\beta_2>0$，但是我们“忘了”把它考虑进去。由于工作时间与朋友的数量是呈正相关的，这就意味着对于朋友数量较多的用户来说，模型给出的预测值往往太小；对于朋友数量较少的用户来说，模型给出的预测值往往太大。这样做的结果是，我们可以通过降低 β_1 的估计值来减少（单变量模型）的误差，即误差最小化的 β_1 是小于其“实际”值的。也就是说，在这种情况下，单变量的最小二乘解偏向于低估 β_1。一般而言，当自变量具有与此类似的误差时，我们的最小二乘解给出的 β 是有偏估计。

15.3 拟合模型

就像对简单线性模型所做的那样，我们这里也需要寻找一个能够最小化误差的平方和的 `beta`。要想以手动方式找到一个精确的解可不是一件容易的事，因此，我们转而求助于梯度下降。下面我们首先创建一个待最小化的误差函数。对于随机梯度下降来说，我们只需要单次预测对应的平方误差：

```
def error(x_i, y_i, beta):
    return y_i - predict(x_i, beta)

def squared_error(x_i, y_i, beta):
    return error(x_i, y_i, beta) ** 2
```

如果你熟悉微积分，可以通过下面的方式进行计算：

```
def squared_error_gradient(x_i, y_i, beta):
    """the gradient (with respect to beta)
    corresponding to the ith squared error term"""
    return [-2 * x_ij * error(x_i, y_i, beta)
            for x_ij in x_i]
```

否则的话，就按照我说的来。

至此，我们就可以利用随机梯度下降法来寻找最优的 beta 了：

```python
def estimate_beta(x, y):
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(squared_error,
                               squared_error_gradient,
                               x, y,
                               beta_initial,
                               0.001)

random.seed(0)
beta = estimate_beta(x, daily_minutes_good) # [30.63, 0.972, -1.868, 0.911]
```

这样的话，我们的模型就变成了：

$$\text{minutes} = 30.63 + 0.972\ \text{friends} - 1.868\ \text{work hours} + 0.911\ \text{phd}$$

15.4 解释模型

你应该把模型的系数看作在其他条件相同的情况下每个因素的影响力的大小。在其他条件相同的情况下，每增加一个朋友，每天花在网站上的时间就会多出一分钟。在其他条件相同的情况下，用户在工作日每多工作一个小时，每天花在网站上的时间就会减少两分钟。在其他条件相同的情况下，拥有博士学位的用户每天用在网站上的时间会多出一分钟。

但是，它没有（直接）反映变量之间的任何相互作用。较之于朋友较少的人而言，工作时间对朋友较多的人的影响很可能是不一样的，而这个模型并没有捕捉到这一点。要想处理这种情况，一种方法是引入一个新变量，即“朋友数量”与“工作时间”之积。这样实际上就使得“工作时间”的系数可以随着朋友数量的增加而增加（或减少）。

还有一种可能，就是朋友越多，花在网站上的时间就越多，但是达到一个上限之后，更多的朋友反而会导致花在网站上的时间变少。（或许是因为朋友太多了反而上网体验会很糟糕？）我们可以设法让模型捕获到这一点，方法是添加另一个变量，即朋友数量的平方。

一旦我们开始添加变量，我们就需要考虑它们的系数“问题”。对于添加的乘积、对数、二次幂以及更高次幂来说，其数量上是没有限制的。

15.5 拟合优度

现在，我们再来看看 R 的平方值，目前已经升至 0.68 了：

```python
def multiple_r_squared(x, y, beta):
    sum_of_squared_errors = sum(error(x_i, y_i, beta) ** 2
                                for x_i, y_i in zip(x, y))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(y)
```

但是不要忘了，只要向回归模型中添加新的变量就必然导致 R 的平方变大。归根结底，前面的简单回归模型只是这里的多重回归模型的特例而已，即“工作时间”和“博士”这两

列的系数都等于 0。因此，最优的多元回归模型，其误差一定不会高于简单回归模型。

因此，对于多重回归分析而言，我们还要考察系数的*标准误差*，即衡量每个 β_i 的估计值的可靠程度。

总的来说，回归模型通常能够很好地拟合我们的数据，但是，如果某些自变量是相关的（或不相关的），那么其系数就*未必*有多大的意义了。

对于这些误差，传统的度量方法通常都带有一个前提假设，即误差 ε_i 是独立的正态随机变量，其平均值为 0，标准偏差为 σ（未知）。那样的话，我们（或者说我们的统计软件）就可以使用线性代数来确定每个系数的标准误差了。这个误差越大，说明模型的系数越不靠谱。令人遗憾的是，我们不打算从零开始介绍这类线性代数。

15.6　题外话：Bootstrap

假设我们有一个含有 n 个数据点的样本，并且这些点是按照某种（我们不知道的）概率分布生成的：

```
data = get_sample(num_points=n)
```

在第 5 章中，我们曾经编写了一个计算观测数据中位数的函数，现在拿它来估算该分布本身的中位数。

但是，我们该如何了解这些估计值的可靠性呢？如果样品中所有的数据都非常接近 100，则实际的中位数很可能也非常接近 100。如果样本中一半左右的数据接近 0，而另一半则接近 200，那么我们就很难确信中位数到底接近多少。

如果我们能够不断获得新的样本，那么就可以计算出每个新样本的中位数，并观察这些中位数的分布情况。但是，一般这是不现实的。相反，我们可以利用 Bootstrap 来获得新的数据集，即选择 n 个数据点并用原来的数据将其*替换*，然后计算合成的数据集的中位数：

```
def bootstrap_sample(data):
    """randomly samples len(data) elements with replacement"""
    return [random.choice(data) for _ in data]

def bootstrap_statistic(data, stats_fn, num_samples):
    """evaluates stats_fn on num_samples bootstrap samples from data"""
    return [stats_fn(bootstrap_sample(data))
            for _ in range(num_samples)]
```

例如，考虑下列两个数据集：

```
# 101个点都非常接近100
close_to_100 = [99.5 + random.random() for _ in range(101)]
```

```
# 101个点钟,50个接近0,50个接近200
far_from_100 = ([99.5 + random.random()] +
                [random.random() for _ in range(50)] +
                [200 + random.random() for _ in range(50)])
```

如果你计算每个数据集的中位数，会发现它们都非常接近 100。然而，如果你考察下面的语句：

```
bootstrap_statistic(close_to_100, median, 100)
```

大部分情况下你看到的数字确实非常接近 100。然而，如果你考察下面的语句：

```
bootstrap_statistic(far_from_100, median, 100)
```

你会发现，不仅有许多数字接近 0，而且还有许多数字接近 200。

第一组中位数的 standard_deviation 接近 0，而第二组中位数的 standard_deviation 接近 100。（这种极端的情况通过人工检查数据很容易弄清楚，但一般情况下都不是真的。）

15.7 回归系数的标准误差

我们可以采用同样的方法来估计回归系数的标准误差。我们可以对数据重复采用 bootstrap_sample 样本，并根据这些样本估算 beta。如果某个自变量（如 num_friends）的系数在各个样本上变化不大，那么就可以确信我们的估计是比较严密的。如果这个系数随着样本的不同而起伏较大，那么我们就不能完全相信我们的估计。

唯一需要说明的是，采样前，我们需要把数据 X 和数据 Y 放到一起（用 zip），以确保对自变量和因变量一起进行采样。这就意味着 bootstrap_sample 将返回一个由 (x_i, y_i) 数据对组成的列表，因此我们需要将其重新组合成一个 x_sample 和一个 y_sample：

```
def estimate_sample_beta(sample):
    """sample is a list of pairs (x_i, y_i)"""
    x_sample, y_sample = zip(*sample) # 魔法般的解压方式
    return estimate_beta(x_sample, y_sample)

random.seed(0) # 所以你得到的结果与我的一样

bootstrap_betas = bootstrap_statistic(zip(x, daily_minutes_good),
                                      estimate_sample_beta,
                                      100)
```

之后，我们就可以估算每个系数的标准偏差了：

```
bootstrap_standard_errors = [
    standard_deviation([beta[i] for beta in bootstrap_betas])
    for i in range(4)]

# [1.174,    # 常数项,        实际误差 = 1.19
```

```
#  0.079,    # num_friends,  实际误差 = 0.080
#  0.131,    # unemployed,   实际误差 = 0.127
#  0.990]    # phd,          实际误差 = 0.998
```

我们可以使用它们来检验诸如“β_i 等于 0 吗？”之类的假设。在满足 β_i=0（以及与 ε_i 分布有关的其他假设）的条件下，则有：

$$t_j = \hat{\beta}_j / \hat{\sigma}_j$$

也就是说，这个统计量等于我们估算的 β_j 除以估算的其标准误差，它符合具有“$n-k$ 个自由度”的学生的 t 分布（Student’s t-distribution）。

如果我们有一个 students_t_cdf 函数，那么就可以计算每个最小二乘系数的 p 值，从而指出实际的系数为 0 时观察到这个值的可能性有多大。令人遗憾的是，实际上我们没有这样的函数。（虽然我们不想从头做起。）

然而，随着自由度变大，t 分布越接近标准正态分布。在这种情况下，即 n 比 k 大得多的情况下，我们便可以使用 normal_cdf 了，并且我们觉得它效果还不错：

```
def p_value(beta_hat_j, sigma_hat_j):
    if beta_hat_j > 0:
        # 如果系数是正的,则我们需要对
        # 看见一个更大的值的概率做两次计算
        return 2 * (1 - normal_cdf(beta_hat_j / sigma_hat_j))
    else:
        # 否则看见一个更小值的概率乘以2
        return 2 * normal_cdf(beta_hat_j / sigma_hat_j)

p_value(30.63, 1.174)    # ~0   (常数项)
p_value(0.972, 0.079)    # ~0   (num_friends)
p_value(-1.868, 0.131)   # ~0   (work_hours)
p_value(0.911, 0.990)    # 0.36 (phd)
```

（在其他情况下，我们很可能会使用一个知道如何计算 t 分布和精确的标准误差的统计软件。）

虽然大多数系数的 p 值都非常小（但非 0 值），但是“博士学位”的系数与零没有“显著”区别，也就是说“博士学位”的系数很可能是随机的，无意义的。

在对回归分析要求更加精细的情形下，你可能需要对数据的各种假设进行更加细致的测试，比如“至少有一个 β_j 是非 0 值”，或者“β_1 等于 β_2 且 β_3 等于 β_4”等，以便进行 F 测试，但是这些内容已经超出了本书的讨论范围。

15.8　正则化

在实践中，线性回归经常需要处理具有很多变量的数据集，这时就需要用到另外两个技

巧。首先，涉及的变量越多，模型越容易对训练集产生过拟合现象。其次，非零系数越多，越难以搞清楚它们的意义。如果我们的目标是*解释*某些现象，一个只考虑三方面因素的稀疏型模型通常要比涉及数百个因素的模型要更好一些。

*正则化*是指给误差项添加一个惩罚项，并且该惩罚项会随着 beta 的增大而增大。然后，我们开始设法将误差项和惩罚项的组合值最小化。因此，惩罚项越大，就越能防止系数过大。

例如，在*岭回归*（ridge regression）中，我们添加了一个与 beta_i 的平方之和成正比的惩罚项。（当然，我们一般不会惩罚 beta_0，因为它是个常数项。）

```
# alpha是一个*超参数*,用来控制惩罚的程度
# 它有时被叫作"lambda",但这在Python中另有所指
def ridge_penalty(beta, alpha):
  return alpha * dot(beta[1:], beta[1:])

def squared_error_ridge(x_i, y_i, beta, alpha):
    """estimate error plus ridge penalty on beta"""
    return error(x_i, y_i, beta) ** 2 + ridge_penalty(beta, alpha)
```

之后你可以按通常的方法插入梯度下降：

```
def ridge_penalty_gradient(beta, alpha):
    """gradient of just the ridge penalty"""
    return [0] + [2 * alpha * beta_j for beta_j in beta[1:]]

def squared_error_ridge_gradient(x_i, y_i, beta, alpha):
    """the gradient corresponding to the ith squared error term
    including the ridge penalty"""
    return vector_add(squared_error_gradient(x_i, y_i, beta),
                      ridge_penalty_gradient(beta, alpha))

def estimate_beta_ridge(x, y, alpha):
    """use gradient descent to fit a ridge regression
    with penalty alpha"""
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(partial(squared_error_ridge, alpha=alpha),
                               partial(squared_error_ridge_gradient,
                                       alpha=alpha),
                               x, y,
                               beta_initial,
                               0.001)
```

如果令 alpha 为 0，则根本不会实施任何惩罚，这时得到的结果跟前面一样：

```
random.seed(0)
beta_0 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.0)
# [30.6, 0.97, -1.87, 0.91]
dot(beta_0[1:], beta_0[1:]) # 5.26
multiple_r_squared(x, daily_minutes_good, beta_0) # 0.680
```

随着 alpha 的增大，拟合优度会变差，但是 beta 会变小：

```
beta_0_01 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.01)
# [30.6, 0.97, -1.86, 0.89]
dot(beta_0_01[1:], beta_0_01[1:])  # 5.19
multiple_r_squared(x, daily_minutes_good, beta_0_01)  # 0.680

beta_0_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.1)
# [30.8, 0.95, -1.84, 0.54]
dot(beta_0_1[1:], beta_0_1[1:])  # 4.60
multiple_r_squared(x, daily_minutes_good, beta_0_1)  # 0.680

beta_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=1)
# [30.7, 0.90, -1.69, 0.085]
dot(beta_1[1:], beta_1[1:])  # 3.69
multiple_r_squared(x, daily_minutes_good, beta_1)  # 0.676

beta_10 = estimate_beta_ridge(x, daily_minutes_good, alpha=10)
# [28.3, 0.72, -0.91, -0.017]
dot(beta_10[1:], beta_10[1:])  # 1.36
multiple_r_squared(x, daily_minutes_good, beta_10)  # 0.573
```

特别地，随着惩罚项的增大，“博士学位”的系数会变成 0，这与我们之前的结果是一致的，即它与 0 没有显著区别。

在利用这个方法之前，通常需要调整数据的规模。因为即使是同一个模型，如果将几年数据一下变为几百年的数据，那么它的最小二乘法系数就会增加上百倍，那样得到的惩罚肯定也会骤增。

还有一个方法是 lasso 回归，它用的惩罚方式如下所示：

```
def lasso_penalty(beta, alpha):
    return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

总的说来，岭回归的惩罚项会缩小系数，但是，lasso 的惩罚项却趋向于迫使系数变为 0 值，这使得它更适于学习稀疏模型。令人遗憾的是，它不适用于梯度下降法，这意味着我们将无法从头开始解决这个问题。

15.9　延伸学习

- 回归分析具有深厚而广阔的理论背景。要想了解这些背景理论，你需要阅读相应的教科书，至少也得阅读大量的维基百科文章。
- scikit-learn 的 linear_model 模块（http://scikit-learn.org/stable/modules/linear_model.html）提供了一个 `LinearRegression` 模型，它跟我们的模型颇为相近。此外，它还提供了 `Ridge` 回归和 `Lasso` 回归，以及其他类型的正则化算法。
- 另一个相关的 Python 模块是 Statsmodels（http://statsmodels.sourceforge.net/），它也包含了线性回归模型及许多其他内容。

第16章

逻辑回归

许多人说，天才和疯子之间只有一线之隔。但是我却认为，那不是一线之隔，而是天渊之别。

——比尔 · 贝利

在第 1 章中，我们简单介绍了如何预测哪些 DataSciencester 用户会成为付费用户。下面，我们将进一步考察这个问题。

16.1　问题

我们有一个 200 人的匿名数据集，内容包括每个用户的工资、作为数据科学家的工作年限以及是否愿意成为付费用户（图 16-1）。像往常一样，对于类别型变量，它们的取值要么是 0（非付费用户），要么是 1（付费用户）。

像往常一样，我们的数据都是存放到矩阵中的，其中每一行都是一个列表 [experience, salary, paid_account]。下面，我们将其转换成所需要的格式：

```
x = [[1] + row[:2] for row in data]   # 每个元素都是[1, experience, salary]
y = [row[2] for row in data]          # 每个元素都是一个付费用户
```

很明显，第一步是使用线性回归来寻找最佳模型：

$$\text{paid account} = \beta_0 + \beta_1 \text{experience} + \beta_2 \text{salary} + \varepsilon$$

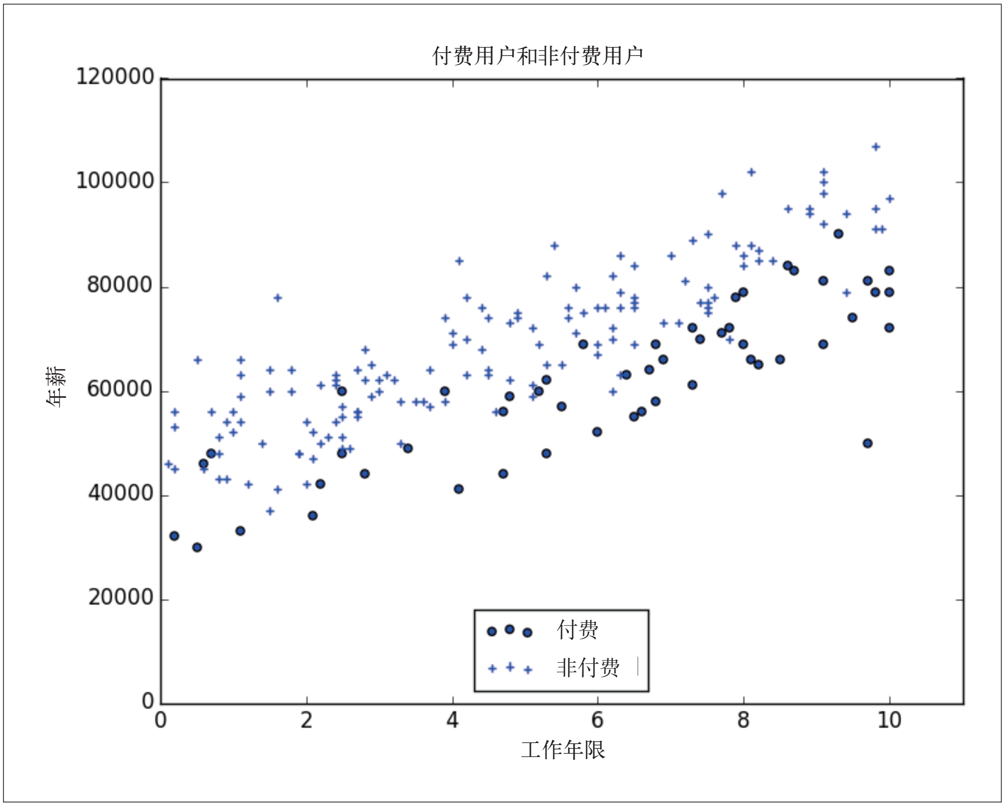


图 16-1：付费用户和非付费用户

当然，在利用这种方式对这个问题进行建模方面，我们已经轻车熟路了。

结果如图 16-2 所示：

```
rescaled_x = rescale(x)
beta = estimate_beta(rescaled_x, y)  # [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_x]

plt.scatter(predictions, y)
plt.xlabel("predicted")
plt.ylabel("actual")
plt.show()
```

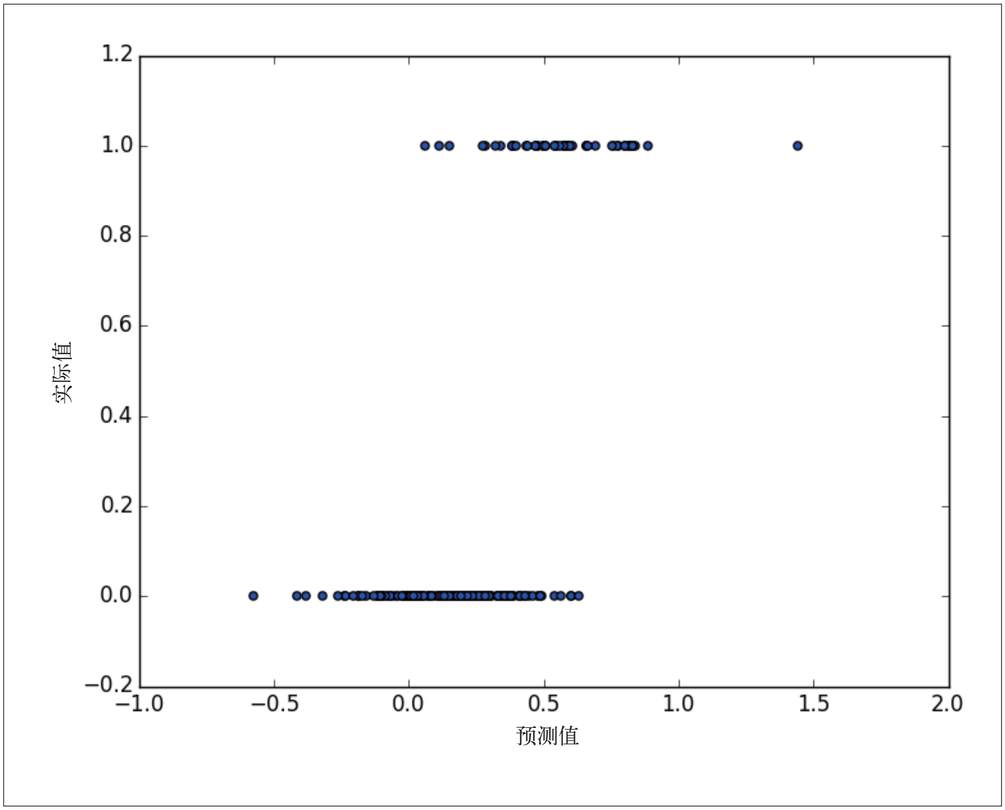


图 16-2：使用线性回归预测付费用户

但这种方法会导致两个紧密相关的问题。

- 我们希望输出结果为 0 或者 1，用来表明会员资格类型。如果输出值介于 0 和 1 之间的话，那也很好，因为我们可以将其解读为概率——如果输出值是 0.25，可以表示成为付费会员的可能性为 25%。但是，线性模型的输出值有时候会是非常大的正数乃至负数，这时候就不好解释了。实际上，这个例子中许多预测值是负数。
- 线性回归模型假定 x 的各列与误差不相关。但是这里 `experience` 一列的回归系数为 0.43，也就是说，数据科学家生涯越长，就越有可能成为付费用户。这意味着对于职业生涯较长的人，模型会输出一个很大的数值。但是我们都知道，有效值最大只能到 1，也就是说输出值越大(即数据科学家生涯越长)，对应的误差项的负值也就越大。由于这个原因，我们对 `beta` 的估计是有偏的。

相反，我们期望的情况是这样的：如果 `dot(x_i, beta)` 的输出值是较大的正数，那么让它对应的概率接近 1；如果输出值是一个较大的负数，那么让它对应的概率接近 0。为此，我们可以应用另一个函数来实现这种效果。

16.2 Logistic函数

对于逻辑回归来说，我们需要用到 Logistic 函数，其图像如图 16-3 所示：

```
def logistic(x):
    return 1.0 / (1 + math.exp(-x))
```

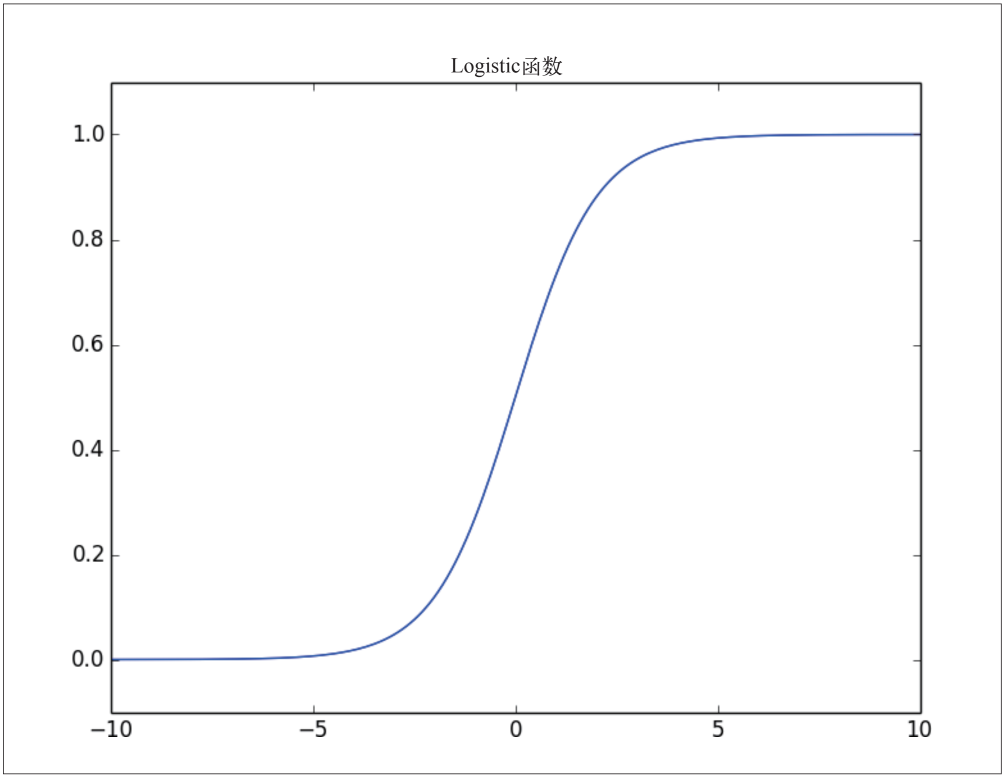


图 16-3：Logistic 函数

随着输入的数字变大且符号为正，它的输出就会越来越接近 1。随着输入的数字变大且符号为负，它的输出就会越来越接近 0。此外，这个函数还有一个非常好的属性，即其导数可以通过下列代码简单求出：

```
def logistic_prime(x):
    return logistic(x) * (1 - logistic(x))
```

这一点可以用于拟合模型：

$$y_i = f(x_i\beta) + \varepsilon_i$$

其中，f 表示 `logistic` 函数。

回想一下，对于线性回归来说，我们是通过最小化误差的平方之和的方式来拟合模型，最终选出令得到这些观测数据的可能性最大的 β。

但是，这里两者并不是等价的，所以我们直接使用梯度下降法来最大化似然。也就是说，我们需要计算似然函数及其梯度。

已知 β，我们的模型指出每个 y_i 等于 1 的概率为 $f(x_i\beta)$，等于 0 的概率为 $1-f(x_i\beta)$。

特别是，y_i 的概率密度函数为：

$$p(y_i \mid x_i, \beta) = f(x_i\beta)^{y_i}(1-f(x_i\beta))^{1-y_i}$$

如果 y_i 为 0，则这等同于：

$$1-f(x_i\beta)$$

且如果 y_i 为 1，则等同于：

$$f(x_i\beta)$$

事实表明，最大化对数似然要更加简单一些：

$$\log L(\beta \mid x_i, y_i) = y_i \log f(x_i\beta) + (1-y_i)\log(1-f(x_i\beta))$$

由于对数函数是单调递增函数，所以任何能够最大化对数似然函数的 beta 必然也能最大化似然函数，反之亦然。

```
def logistic_log_likelihood_i(x_i, y_i, beta):
    if y_i == 1:
        return math.log(logistic(dot(x_i, beta)))
    else:
        return math.log(1 - logistic(dot(x_i, beta)))
```

如果我们假设各个数据点之间相互独立，那么整体的似然就是各个似然之积。换句话说，整体的对数似然就是各个对数似然之和：

```
def logistic_log_likelihood(x, y, beta):
    return sum(logistic_log_likelihood_i(x_i, y_i, beta)
               for x_i, y_i in zip(x, y))
```

利用少许微积分知识，我们就能求出梯度了：

```
def logistic_log_partial_ij(x_i, y_i, beta, j):
    """here i is the index of the data point,
    j the index of the derivative"""

    return (y_i - logistic(dot(x_i, beta))) * x_i[j]

def logistic_log_gradient_i(x_i, y_i, beta):
```

```
    """the gradient of the log likelihood
    corresponding to the ith data point"""

    return [logistic_log_partial_ij(x_i, y_i, beta, j)
            for j, _ in enumerate(beta)]

def logistic_log_gradient(x, y, beta):
    return reduce(vector_add,
                  [logistic_log_gradient_i(x_i, y_i, beta)
                   for x_i, y_i in zip(x,y)])
```

好了，到此为止我们已经万事俱备了。

16.3　应用模型

现在我们要将数据分为一个训练集和一个测试集：

```
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_x, y, 0.33)

# 希望在训练数据集上最大化对数似然
fn = partial(logistic_log_likelihood, x_train, y_train)
gradient_fn = partial(logistic_log_gradient, x_train, y_train)

# 选取一个随机起始点
beta_0 = [random.random() for _ in range(3)]

# 使用梯度下降法实现最大化
beta_hat = maximize_batch(fn, gradient_fn, beta_0)
```

另外，你也可以使用随机梯度下降法：

```
beta_hat = maximize_stochastic(logistic_log_likelihood_i,
                               logistic_log_gradient_i,
                               x_train, y_train, beta_0)
```

无论使用哪种方式，我们都能得到大致如下的结果：

```
beta_hat = [-1.90, 4.05, -3.87]
```

这些数据是按照某些系数转换过来的，不过，我们还可以将其转换为原始数据：

```
beta_hat_unscaled = [7.61, 1.42, -0.000249]
```

令人遗憾的是，这些系数不如线性回归系数那样易于解释。在其他条件都相同的情况下，工作年限每增加一年，`logistic` 的输入就会增加 1.42。在其他条件都相同的情况下，年薪每增加 10 000 美元，`logistic` 的输入就会减去 2.49。

然而，输出的结果还会受到其他输入数据的影响。如果 `dot (beta, x_i)` 的值已经很大了（相当于概率接近 1），那么即使再增加的话，对概率也没有多大的影响了。如果它接近 0，

那么即使稍微增加一点，对概率也会产生明显的影响。

也就是说，在其他条件相同的情况下，工作年限越多的人越有可能成为付费用户。同时，其他条件相同的情况下，年薪越高的人越不可能成为付费用户。（当我们将数据绘成图表的时候，这一点就会更明显。）

16.4　拟合优度

目前为止，我们还没有使用留出来的测试数据。下面来看看，如果我们预测成为付费用户的概率大于 0.5 的话会发生什么：

```
true_positives = false_positives = true_negatives = false_negatives = 0

for x_i, y_i in zip(x_test, y_test):
    predict = logistic(dot(beta_hat, x_i))

    if y_i == 1 and predict >= 0.5:    # TP: 是付费用户,且我们预测为是
        true_positives += 1
    elif y_i == 1:                     # FN: 是付费用户,且我们预测为否
        false_negatives += 1
    elif predict >= 0.5:               # FP: 非付费用户,且我们预测为是
        false_positives += 1
    else:                              # TN: 非付费用户,且我们预测为否
        true_negatives += 1

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
```

这里的查准率为 93%（即每预测 100 次有 93 次是正确的），查全率为 82%（即每 100 个付费用户中，我们能够预测出 82 个人），这两项指标都相当不错。

在图 16-4 中，我们给出了系统的预测值与实际值的比较情况，图中表明该模型的表现非常好：

```
predictions = [logistic(dot(beta_hat, x_i)) for x_i in x_test]
plt.scatter(predictions, y_test)
plt.xlabel("predicted probability")
plt.ylabel("actual outcome")
plt.title("Logistic Regression Predicted vs. Actual")
plt.show()
```

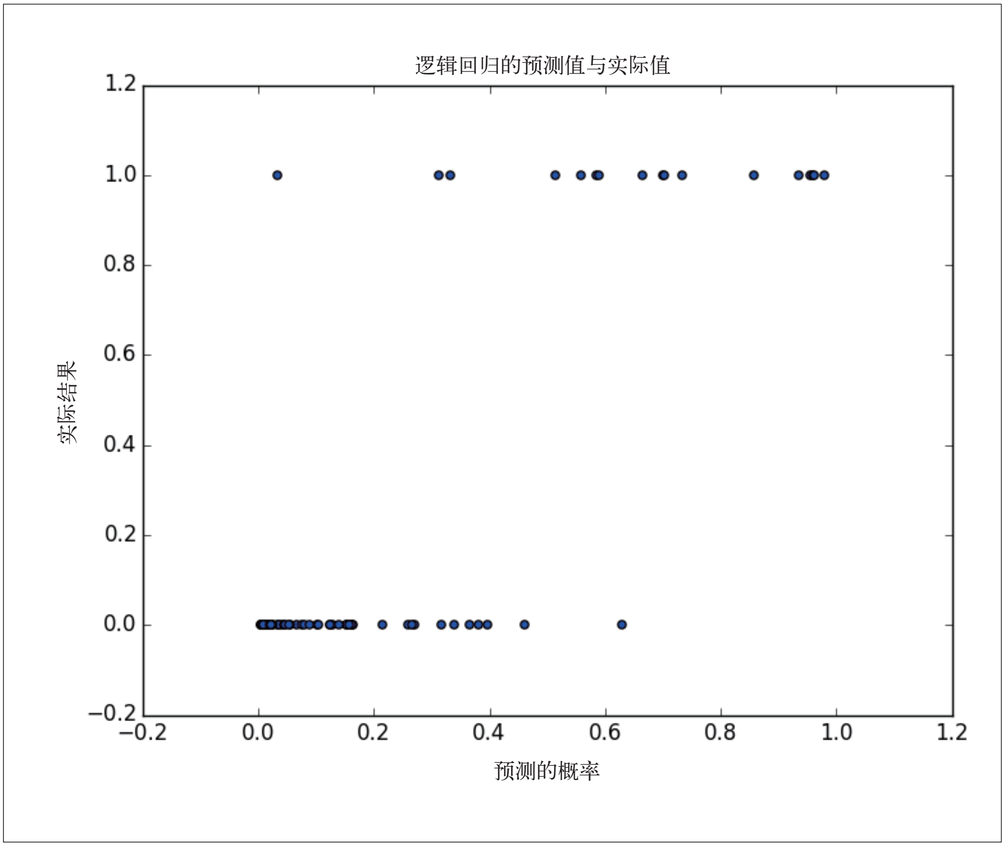


图 16-4：逻辑回归的预测值与实际值

16.5 支持向量机

`Dot(beta_hat, x_i)` 等于 0 的点就是我们的分类边界线，具体如图 16-5 所示。

这个边界实际上就是一个*超平面*（hyperplane），将参数空间一分为二，一半对应着*预测为付费用户*，一半对应着*预测为非付费用户*。我们发现，这个预测只是寻找最优逻辑模型过程中的一个副产品而已。

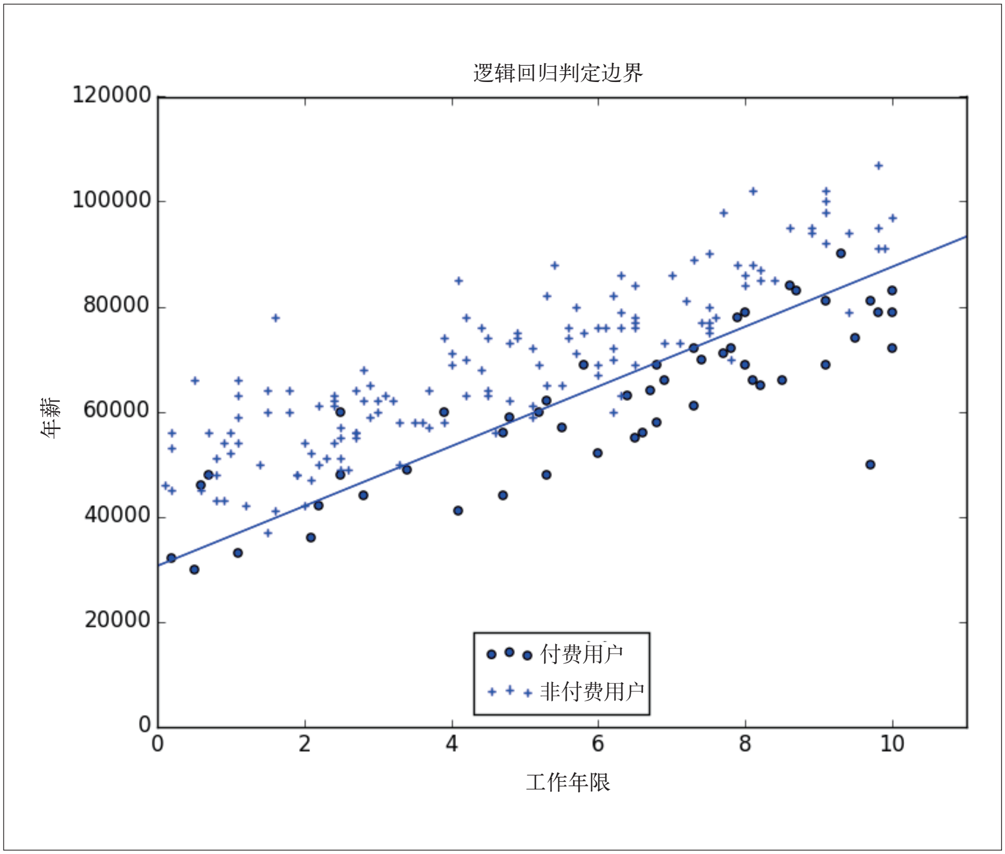


图 16-5：付费用户和非付费用户的判定边界

另外，还有一种分类方法，即寻找的超平面只要对训练数据的分类效果“最佳”即可。这实际上就是支持向量机（support vector machine）思想，即寻找将距离每个类别中的最近点的距离最大化的超平面，如图 16-6 所示。

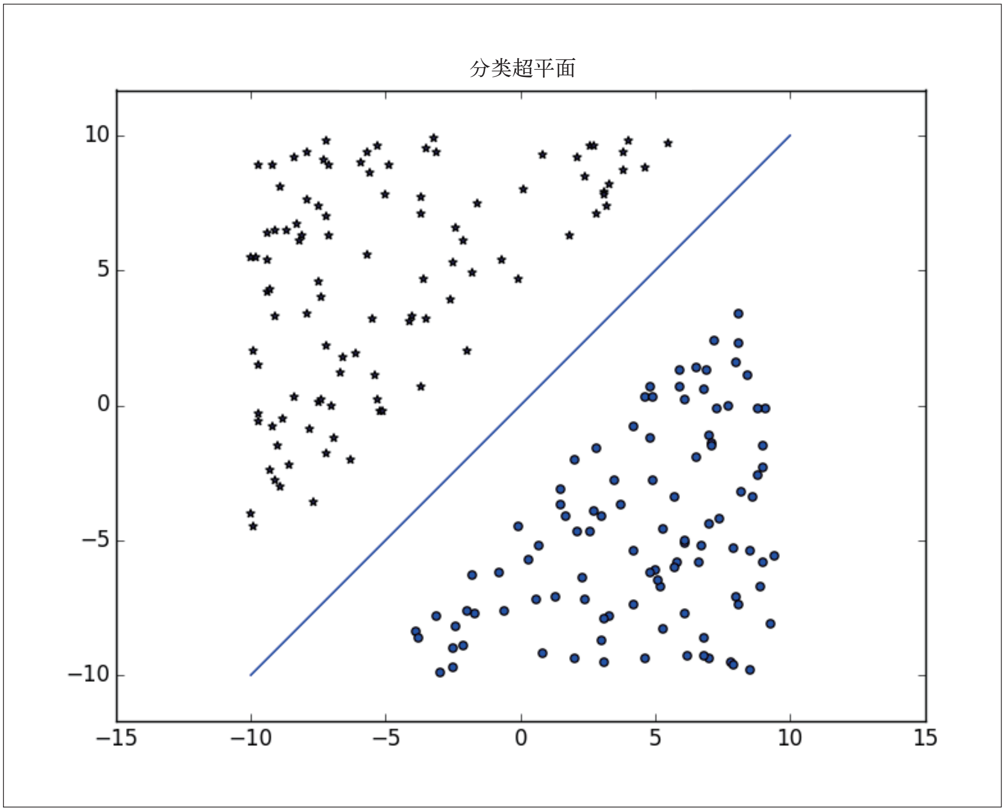


图 16-6：分类超平面

寻找这种超平面的过程就是一个最优化的过程，不过这里面所涉及的技术对我们来说太复杂了。另外一个不同的问题是，分类超平面也许根本就不存在。简单来说，就是在我们的“谁会付费？”的数据集中，没有能够把付费用户和非付费用户完美分隔的直线。

我们（有时）可以考虑把数据映射到一个更高维的空间中。例如，我们先看图 16-7 所示的一维数据集的情形。

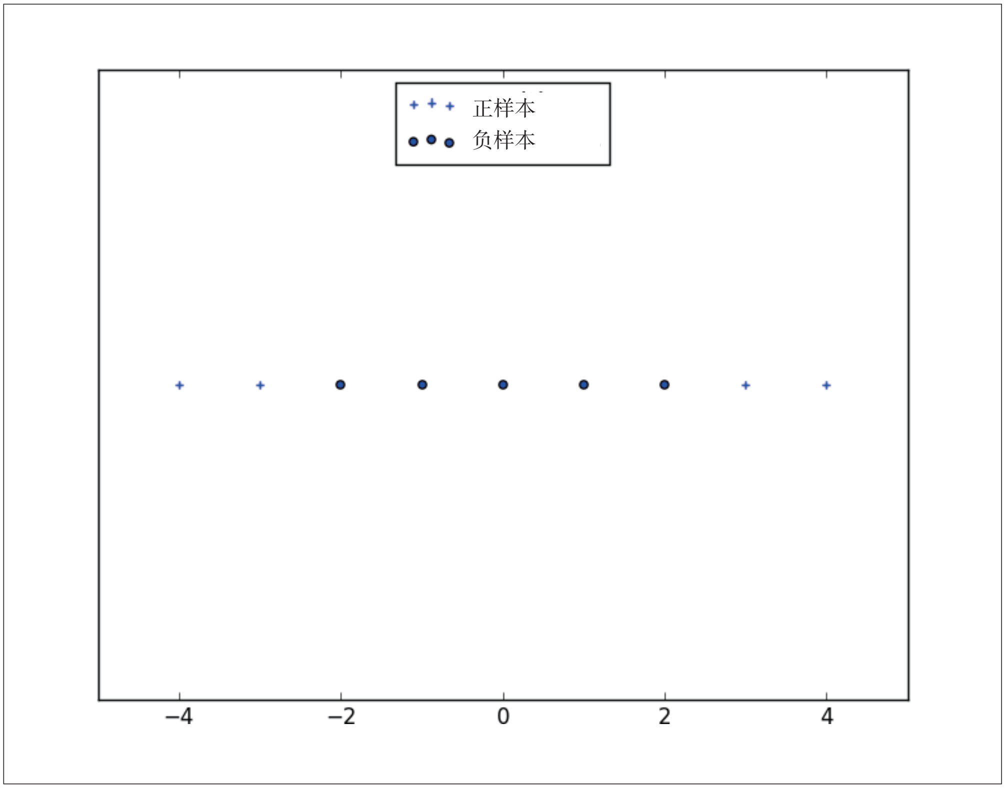


图 16-7：无法分隔的一维空间数据集

很明显，没有一个超平面能够将这些正样本与负样本分隔开来。但是，如果通过把 x 替换为 (x, x**2) 来将数据集映射到一个二维空间，我们看一下情况会有什么变化。情况突然出现了转机：我们能够找出分隔数据的超平面了，如图 16-8 所示。

这通常称为核方法（kernel trick），因为不用真的把数据点映射到更高维的空间（如果数据数量较多并且映射复杂的话，这个过程的代价将会很大），相反，我们可以使用“核”函数来计算更高维空间中的点积，并用它们来找出超平面。

如果不借助于专业人士编写的专门的优化软件的话，我们会很难（并且也不一定是一个好主意）利用支持向量机，因此，我们对它的介绍到此为止。

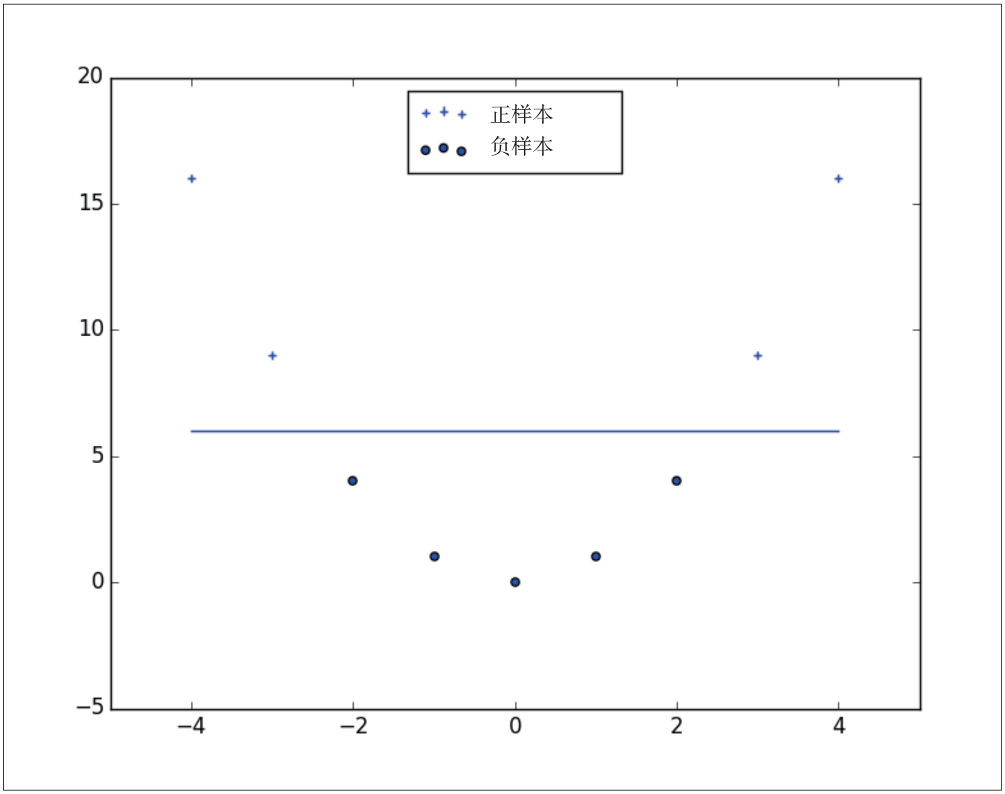


图 16-8：在更高维空间中数据变为可分隔的

16.6 延伸学习

- scikit-learn 库同时提供了逻辑回归（http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression）和支持向量机（http://scikit-learn.org/stable/modules/svm.html）的相关模块。
- scikit-learn 实际上是利用 libsvm（http://www.csie.ntu.edu.tw/~cjlin/libsvm/）来实现的支持向量机。其网站上提供了许多支持向量机方面的优秀资料。

第17章
决策树

> 每一棵树都是一个解不开的谜。
>
> ——吉姆·伍德林

假设 DataSciencester 人力副总通过网站召集了许多求职者，并面试了他们，当然，对他们的满意程度各不相同。他还收集了一组数据，其中包括每个求职者的各种（定性的）特质以及面试表现情况。现在他向你咨询：能否用这些数据建立一个模型，从而识别出哪些应聘者能够通过面试？如果可以的话，那就不必浪费时间进行面试了。

这个问题看起来非常适合利用*决策树*（decision tree）来解决。所谓决策树，是数据科学家工具箱中的另一种预测建模工具。

17.1 什么是决策树

决策树通过树结构来表示各种可能的*决策路径*（decision path），以及每个路径的结果。

如果你之前玩过二十问（Twenty Questions，https://en.wikipedia.org/wiki/Twenty_Questions）这个游戏的话，那么你早就熟悉决策树了，举例来说：

- “我在猜一种动物。”
- “它有五条以上的腿吗？”
- “否。”
- “好吃吗？”
- “否。”

- “是否出现在澳大利亚五分硬币的背面？”
- “是。”
- “是针鼹吗？”
- “对，正是！”

下面是相应的判断路径。

“不超过 5 条腿”→“不好吃”→“出现在 5 分硬币上”→“针鼹！”这就是一棵特殊的（而不是非常宽泛的）“猜动物”的决策树，如图 17-1 所示。

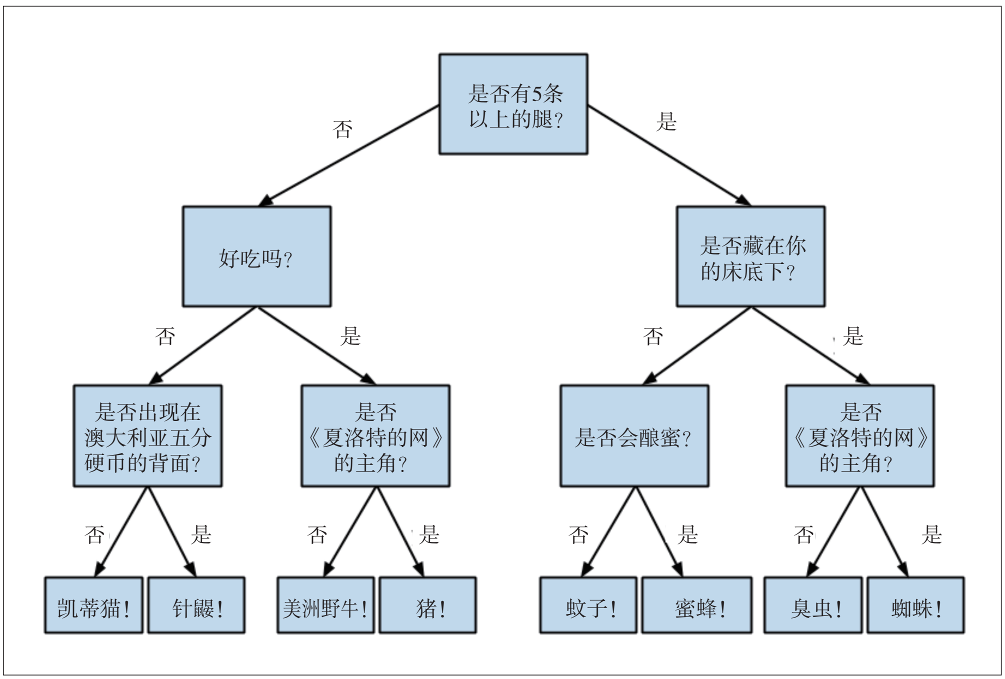


图 17-1：“猜动物”决策树

决策树不仅能够提供很多建议，并且非常易于理解和解释，同时，进行推断的过程还是完全透明的。与本书前面介绍的模型不同，决策树不仅可以轻松处理混合在一起的数值属性（例如腿数）和条件属性（例如好吃 / 不好吃），甚至还可以对缺失属性的数据进行分类。

不过，要想利用一组训练数据找出“最优”决策树却是一个非常艰巨的计算问题。（考虑到计算量的问题，我们这里要建立的是一棵足够好的决策树，而非最优决策树。即便如此，当数据集变大时，计算量仍旧会面临挑战。）更重要的是，建立决策树模型时非常容易出现对训练数据的严重过拟合现象，从而导致模型对于未曾见过的数据的效果大打折扣。关于这个问题，我们将设法加以解决。

大多数人都将决策树分为分类决策树（classification tree，它输出的是判决结果）和回归决策树（regression tree，它输出的是数值结果）。本章我们将重点介绍分类决策树，同时还通过 ID3 算法根据已标记的数据集来确定决策树，这将有助于我们理解决策树的实际运行机制。为简单起见，我们只探讨仅有两个输出结果的问题，如“我该不该雇用这个应聘者？”“我应该给这个网站访问者展示广告 A 还是广告 B？”或者“吃了从办公室冰箱里发现的这些食物会不会反胃？”

17.2　熵

为了建立一个决策树，我们需要决定提出哪些问题，以及这些问题的提问顺序。树的每个阶段都存在一些不确定性，其中有些不确定性已经被我们消除，而另一些则依旧存在。当知道该动物的腿不超过五条后，我们就已经排除了它是蝗虫的可能性。但是，这并没有排除它是鸭子的可能性。对于每一个可能的问题，我们都可以根据其答案对剩余的可能性空间做进一步分割。

理想情况下，我们当然愿意选择那些具有能够给决策树的预测提供更多信息的答案的问题。如果有一个是 / 否问题，并且答案为“是”的时候输出为 `True`，而答案为“否”的时候输出为 `False`（反之亦然），那么这样的问题自然是我们的首选了。相反，如果一个是 / 否问题的答案无论是啥都不能为预测提供新信息的话，那么它就不是一个好的选择。

我们用熵（entropy）这个概念来指代“信息含量”，此外，这个词还常用来表示混乱程度。在这里，我们用它来表示与数据相关的不确定性。

假设我们有一个数据集 S，每个数据元素都标明了所属的类别，即元素属于有限类别 $C_1,\cdots, C_n$ 中的一种。如果所有数据点都属于同一类别，那么也就不存在不确定性了，这就属于我们喜闻乐见的低熵情形。如果数据点均匀地分布在各个类别中，那么不确定性就较大，这时我们说具有较大的熵。

从数学的角度来讲，如果 p_i 表示 c_i 类别中的数据所占的比例，那么可以把熵定义为：

$$H(S)=-p_1\log_2 p_1-\cdots-p_n\log_2 p_n$$

按照通常的约定，$0\log 0=0$。

对于这个定义，我们不必关心其中的细枝末节，只要明白每一个 $-p_i\log_2 p_i$ 项都是非负的，并且当 p_i 接近 0 或 1 时，熵的值也接近 0（如同图 17-2 所示）即可。

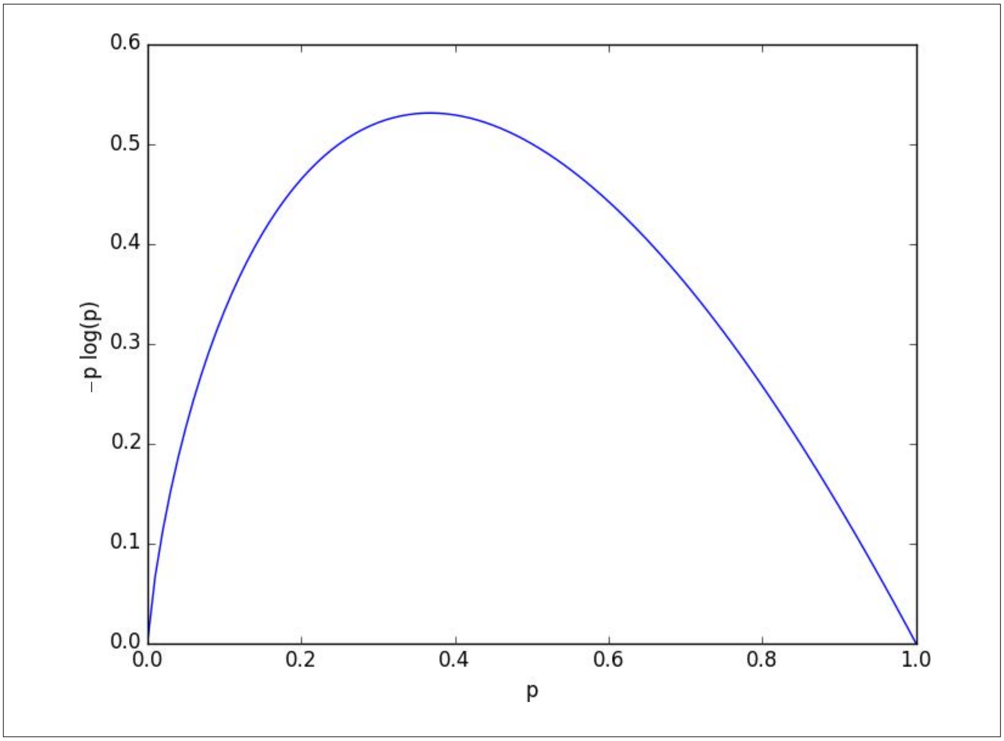


图 17-2：$-p\log p$ 的图像

这就意味着，当每一个 p_i 越接近 0 或 1 时（即当大部分数据都属于同一个类别时），熵就越小；当许多 p_i 不接近 0 时（即当数据广泛分布于众多类别中时），熵就越大。这正是我们所期望的特性。

我们可以轻而易举地将上面的定义编写为一个函数：

```
def entropy(class_probabilities):
    """given a list of class probabilities, compute the entropy"""
    return sum(-p * math.log(p, 2)
               for p in class_probabilities
               if p)                          # 忽略零可能性
```

我们的数据点是由一对 (input, label) 组成的，这就意味着类别概率需要我们自己来计算。需要注意的是，我们并不关心标签与概率之间的关联，我们只在乎概率本身：

```
def class_probabilities(labels):
    total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
```

```
    probabilities = class_probabilities(labels)
    return entropy(probabilities)
```

17.3　分割之熵

迄今为止，我们所做的是计算单组标记数据的熵（即“不确定性”）。实际上，决策树每前进一步，都要提出一个问题，而它的答案会把数据分割为一个或（更可能的情况是）多个子集。例如，“是否有五条以上的腿？”这个问题能够把动物分为五条腿以上的（如蜘蛛）和非五条腿以上的（如针鼹）。

相应地，我们希望通过某种方法对数据集的分割效果来表示熵。对于某个划分方法，如果得到的子集的熵较低（即确定性很高）的话，我们就说这个划分方法的熵较低；反之，如果得到的子集的（数量很多并且）熵较高（即不确定性很高）的话，我们就说这个划分方法的熵较高。

例如，前面“澳大利亚五分硬币”就是一个非常不明智（尽管这次非常幸运！）的问题，因为它把剩余的动物分为 S_1 ={ 针鼹 } 和 S_2={ 针鼹之外的一切动物 } 两个子集，而 S_2 这个集合不仅过大而且高熵。（虽然 S_1 子集没有熵，但它只能代表剩余“类别”中的一小部分。）

在数学上，如果我们把数据集 S 划分为数据子集 $S_1,\cdots, S_m$，各个子集相应数据量所占比例为 $q_1,\cdots, q_m$，那么我们就可以通过如下加权和的形式来计算这次划分的熵：

$$H=q_1H(S_1)+\cdots+q_mH(S_m)$$

下面是具体的实现代码：

```
def partition_entropy(subsets):
    """find the entropy from this partition of data into subsets
    subsets is a list of lists of labeled data"""

    total_count = sum(len(subset) for subset in subsets)

    return sum( data_entropy(subset) * len(subset) / total_count
                for subset in subsets )
```

这种方法的一个问题是，通过具有许多不同的值的属性进行数据划分的话，往往会由于过度拟合而导致过低的熵。例如，假设你为一家银行工作，并尝试用一些历史数据作为训练集，建立一个决策树来预测哪些客户很可能拖欠抵押贷款。我们进一步假设数据集提供了每个客户的社会保险号（SSN）。如果利用 SSN 号码对数据集进行划分，那么得到的每个子集都只含有一个人员，这样的话它们的熵必定为 0。但是，这个依赖于 SSN 号码的模型不一定适用于该训练集之外的数据。出于这个原因，你应该尽量避免使用（或直接去掉，如果合适的话）有大量的可能取值的属性来创建决策树。

17.4 创建决策树

副总为你提供了应聘者的相关资料，包括符合你的数据规范的 (input，label) 对，其中每个输入都是由应聘者的各种属性构成的一个字典变量，而每个标签的取值要么为 True（该求职者面试成绩很好），要么为 False（该求职者面试成绩很差）。具体来说，数据为你提供了每个求职者的得分情况、常用语言、在 Twitter 上的活跃程度以及是否拥有博士学位等信息：

```
inputs = [
    ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'no'},    False),
    ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'yes'},   False),
    ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'no'},      True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'no'},   True),
    ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'yes'},     False),
    ({'level':'Mid', 'lang':'R', 'tweets':'yes', 'phd':'yes'},         True),
    ({'level':'Senior', 'lang':'Python', 'tweets':'no', 'phd':'no'},  False),
    ({'level':'Senior', 'lang':'R', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'yes', 'phd':'no'},  True),
    ({'level':'Senior', 'lang':'Python', 'tweets':'yes', 'phd':'yes'}, True),
    ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'yes'},     True),
    ({'level':'Mid', 'lang':'Java', 'tweets':'yes', 'phd':'no'},       True),
    ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'yes'}, False)
]
```

我们的决策树含有许多*决策节点*（该节点会提出一个问题，并根据问题的答案来指导我们下一步如何走）和*叶节点*（该节点为我们提供预测结果）。我们将使用较为简单的 ID3 算法来创建决策树，具体过程将在下面详细介绍。假设我们得到了一些标记过的数据，以及一个用来选择下一个分支的属性列表。

- 如果所有数据都有相同的标签，那么创建一个预测最终结果即为该标签所示的叶节点，然后停止。
- 如果属性列表是空的（即已经没有更多的问题可提问了），就创建一个预测结果为最常见的标签的叶节点，然后停止。
- 否则，尝试用每个属性对数据进行划分。
- 选择具有最低划分熵的那次划分的结果。
- 根据选定的属性添加一个决策节点。
- 针对划分得到的每个子集，利用剩余属性重复上述过程。

这就是所谓的“贪婪”算法，因为在每一步，它都会选择最快最好的那一个。对于特定的数据集，有的决策树在开头几步看起来表现不佳，但最终结果却可能是最棒的。如果是这样的话，这个算法就无法找到这样的决策树。尽管如此，它也有自己的优点，即相对来说还是很容易理解和实现的，所以，把它用作探索决策树的起点还是非常不错的。

下面让我们以手动方式在应聘者数据集上完成这些步骤。这个数据集具有 True 和 False 两种标签，我们将利用 4 个属性对其进行分类。因此，我们首先要做的就是找出熵最小的分割方法。我们将通过如下函数来完成分割：

```
def partition_by(inputs, attribute):
    """each input is a pair (attribute_dict, label).
    returns a dict : attribute_value -> inputs"""
    groups = defaultdict(list)
    for input in inputs:
        key = input[0][attribute]   # 得到特定属性的值
        groups[key].append(input)   # 然后把这个输入加到正确的列表中
    return groups
```

我们可以通过下列代码来计算熵：

```
def partition_entropy_by(inputs, attribute):
    """computes the entropy corresponding to the given partition"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())
```

然后我们只需要找出在整个数据集上具有最小熵的分割即可：

```
for key in ['level','lang','tweets','phd']:
    print key, partition_entropy_by(inputs, key)

# level 0.693536138896
# lang 0.860131712855
# tweets 0.788450457308
# phd 0.892158928262
```

我们看到，利用 level 进行的分割的熵最小，所以我们需要为每一个可能的 level 值建立一个子树。所有 Mid 应聘者都被标记成了 True，这意味着 Mid 子树是一个叶节点，其预测结果为 True。对于 Senior 级别的求职者，其标签既有 True 也有 False，所以我们需要进一步划分：

```
senior_inputs = [(input, label)
                 for input, label in inputs if input["level"] == "Senior"]

for key in ['lang', 'tweets', 'phd']:
    print key, partition_entropy_by(senior_inputs, key)

# lang 0.4
# tweets 0.0
# phd 0.950977500433
```

这表明，我们下一步应该根据 tweets 进行分割，因为它能得到熵为 0 的分割。对于 Senior 级别的应聘者，tweets 的值为“yes”的最终分类结果为 True，而 tweets 的值为“no”的最终分类结果为 False。

最后，如果我们对 Junior 级别的应聘者做同样的事情，最终会根据属性 phd 进行划分，并且发现没有博士学位的结果都是 True，拥有博士学位的结果都是 False。

图 17-3 为我们展示了完整的决策树。

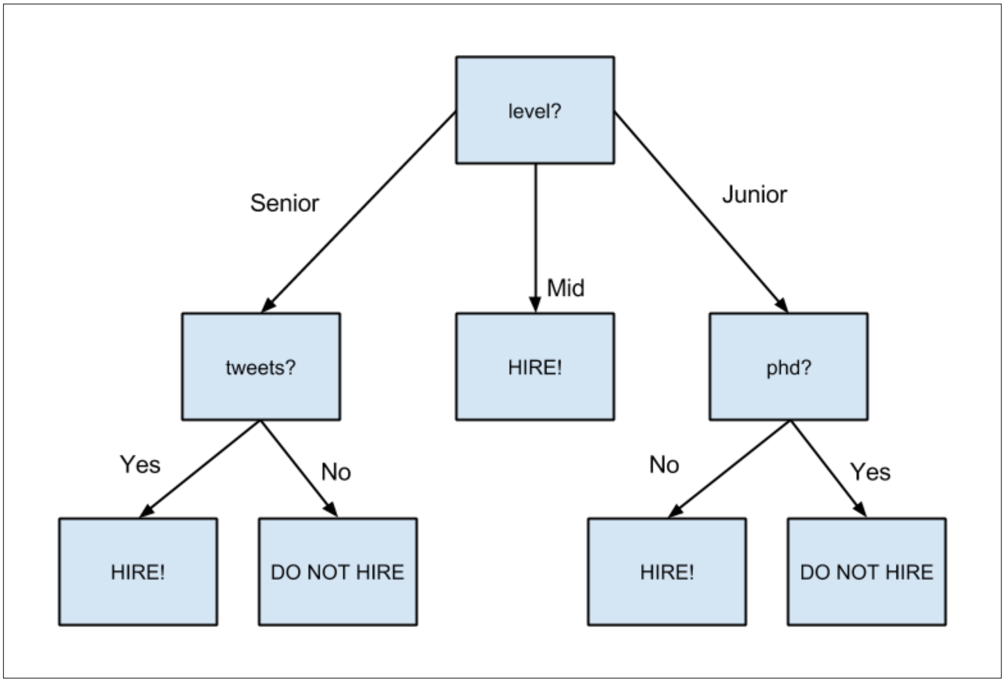


图 17-3：招聘决策树

17.5 综合运用

既然我们已经知道了这个算法的工作原理，下面就以更加通用的方式来实现这个算法。这意味着我们需要决定如何表示决策树。这里，我们将尽可能使用最轻量化的表示方法。我们将树定义为下列情况之一：

- True
- False
- 元组 (attribute, subtree_dict)

这里的 True 代表一个叶节点，对于任何输入该节点都会返回 True；False 也表示一个叶节点，但是对于任何输入该节点都会返回 False；而元组则代表一个决策节点，对于任何输入，该节点都会根据 attribute 的值利用相应的子树对输入进行分类。

使用这种方法，我们的招聘决策树将表示如下：

```
('level',
 {'Junior': ('phd', {'no': True, 'yes': False}),
  'Mid': True,
  'Senior': ('tweets', {'no': False, 'yes': True})})
```

不过还有一个问题需要解决，即如何处理非预期的属性值和缺失属性值的情形。如果招聘决策树遇到应聘者的 level 属性值为“Intern”的情况，该如何处置呢？我们可以通过添加一个关键字 None 来处理这种情况，这时只要把预测结果设为最常见的标签即可。（当然，如果数据集中实际上含有 None 这个值的话，这将是一个糟糕的主意。）

给定了表示方法后，我们就可以对输入进行分类了，具体如下所示：

```
def classify(tree, input):
    """classify the input using the given decision tree"""

    # 如果这是一个叶节点,则返回其值
    if tree in [True, False]:
        return tree

    # 否则这个树就包含一个需要划分的属性
    # 和一个字典,字典的键是那个属性的值
    # 值是下一步需要考虑的子树
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)    # 如果输入的是缺失的属性,则返回None

    if subtree_key not in subtree_dict:   # 如果键没有子树
        subtree_key = None                # 则需要用到None子树

    subtree = subtree_dict[subtree_key]   # 选择恰当的子树
    return classify(subtree, input)       # 并用它来对输入分类
```

最后要做的就是利用训练数据建立决策树的具体表示形式：

```
def build_tree_id3(inputs, split_candidates=None):

    # 如果这是第一步
    # 第一次输入的所有的键就都是split candidates
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # 对输入里的True和False计数
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues

    if num_trues == 0: return False     # 若没有True,则返回一个"False"叶节点
    if num_falses == 0: return True     # 若没有False,则返回一个"True"叶节点

    if not split_candidates:            # 若不再有split candidates
        return num_trues >= num_falses  # 则返回多数叶节点
```

```
    # 否则在最好的属性上进行划分
    best_attribute = min(split_candidates,
                         key=partial(partition_entropy_by, inputs))

    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates
                      if a != best_attribute]

    # 递归地创建子树
    subtrees = { attribute_value : build_tree_id3(subset, new_candidates)
                 for attribute_value, subset in partitions.iteritems() }

    subtrees[None] = num_trues > num_falses      # 默认情况

    return (best_attribute, subtrees)
```

在我们所建的树上，每一个叶节点要么由清一色的 True 输入组成，要不就是由清一色的 False 输入组成。这意味着，该决策树对于这个训练数据集的预测效果堪称完美。但我们也可以把它应用到训练集之外的新数据上面：

```
tree = build_tree_id3(inputs)

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "no"} )        # True

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "yes"} )       # False
```

同时，也可以将它应用于具有缺失值或非预期值的数据：

```
classify(tree, { "level" : "Intern" } ) # True
classify(tree, { "level" : "Senior" } ) # False
```

> 由于我们的目的主要是演示如何构建决策树，因此这里使用了整个数据集来建立决策树。与往常一样，如果现实中我们想创造一个优秀模型的话，就应该（收集更多的数据并且）将数据分成训练子集、验证子集和测试子集。

17.6 随机森林

由于决策树与其训练数据的契合程度非常高，因此，它总是倾向于出现过拟合现象。为了避免出现这种情况，可以使用*随机森林*（random forest）技术。利用该技术，我们可以建立多个决策树，然后通过投票方式决定如何对输入进行分类：

```
def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]
```

我们知道，决策树的构建是一个确定性的过程，那么如何才能得到随机的决策树呢？

总的来说，这需要对数据进行 Bootstrap 抽样处理（这种抽样方法我们曾经在 15.6 节“题外话：Bootstrap”介绍过）。这种方法不是利用训练集合中的所有的输入数据来训练每棵决策树，而是利用 `bootstrap_sample(inputs)` 的取样结果来训练每棵决策树。因为每一棵决策树都是用不同的数据建立的，因此与其他决策树相比，每一棵都有其独特之处。（该方法的另一个好处是可以统一使用非抽样数据来测试每一棵决策树，这意味着如果你的模型效果评测方式设计得巧妙，完全可以将所有数据都用于训练集。）这种技术就是著名的 Bootstrap *集成法*（bootstrap aggregating），或者简称 bagging 方法。

随机性的另一个来源是在分类时不断变换选择最佳属性（`best_attribute`）进行划分的方法。这里不是说选择全部的剩余属性进行划分，而是先从中随机选取一个子集，然后从中寻找最佳属性进行划分：

```
# 如果已经存在了几个足够的划分候选项,就查看全部
if len(split_candidates) <= self.num_split_candidates:
    sampled_split_candidates = split_candidates
# 否则选取一个随机样本
else:
    sampled_split_candidates = random.sample(split_candidates,
                                             self.num_split_candidates)

# 现在仅从这些候选项中选择最佳属性
best_attribute = min(sampled_split_candidates,
    key=partial(partition_entropy_by, inputs))

partitions = partition_by(inputs, best_attribute)
```

上面的代码展示的是一种用途更广泛的技术，称为*集成学习*（ensemble learning），它能够将多个*较弱的*模型（weak learner，通常是高偏差、低方差模型）组合成一个更加强大的模型。

随机森林是最为流行的一种集成方法，由其衍生的模型几乎到处可见。

17.7 延伸学习

- scikit-learn 库提供了许多决策树模型（http://scikit-learn.org/stable/modules/tree.html）。此外，它还提供了一个 ensemble 模块（http://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble），其中包括 `RandomForestClassifier` 以及其他集成方法。
- 我们这里仅仅介绍了决策树及其算法的皮毛知识，如果读者有志于进一步探索该主题，可以先从维基百科（http://en.wikipedia.org/wiki/Decision_tree_learning）的介绍开始学习。

第18章
神经网络

我喜欢胡思乱想，因为这样能唤醒脑细胞。

——苏斯博士

人工神经网络（artificial neural network，或简称神经网络）是受大脑启发而开发出来的一种预测模型。我们可以把大脑看作一团相互连接的神经元。每个神经元都以其他神经元的输出为输入，进行相应计算，如果结果超过某个阈值，则这个神经元将会进入激活状态，否则它会继续保持非激活状态。

相应地，人工神经网络则是由人工神经元组成，同样也对输入进行类似的计算。神经网络可以解决各式各样的问题，比如手写体识别与面部识别等，同时，深度学习作为数据科学最为火爆的一个分支也大量用到神经网络。然而，大部分神经网络都是些“黑盒子”，也就是说，即使考察了其工作细节，也很难获悉它们到底是如何解决问题的。此外，大型的神经网络的训练工作难度也非常大。作为一名处于“发育期”的数据科学家，你所遇到的大多数问题都不适合用神经网络来处理。有朝一日，当你试图打造一个催生奇点的人工智能的时候，或许神经网络会是个不错的选择。

18.1　感知器

感知器（perception）可能是最简单的神经网络了，或者说是由具有 n 个二进制输入的单个神经元所组成的神经网络。感知器首先会对输入值加权求和，如果加权和大于等于 0，它就会被激活：

```
def step_function(x):
    return 1 if x >= 0 else 0

def perceptron_output(weights, bias, x):
    """returns 1 if the perceptron 'fires', 0 if not"""
    calculation = dot(weights, x) + bias
    return step_function(calculation)
```

实际上，感知器只是根据点 x 的超平面将问题空间分隔为两部分而已：

```
dot(weights,x) + bias == 0
```

通过正确选用权值，感知器能解决许多简单的问题（图 18-1）。例如，我们可以创建一个与门（即 AND，也就是说，当两个输入都为 1 时，返回 1；只要输入有一个为 0 时，返回 0），代码如下所示：

```
weights = [2, 2]
bias = -3
```

如果两个输入都为 1，则计算结果为 2 + 2 − 3 = 1，所以输出为 1。但是，只要输入中有一个为 0，则计算结果为 2 + 0 − 3 = -1，所以输出为 0。同时，如果两个输入都为 0，则计算结果为 −3，所以输出还是 0。同样，我们还可以建立一个或门（OR），代码如下所示：

```
weights = [2, 2]
bias = -1
```

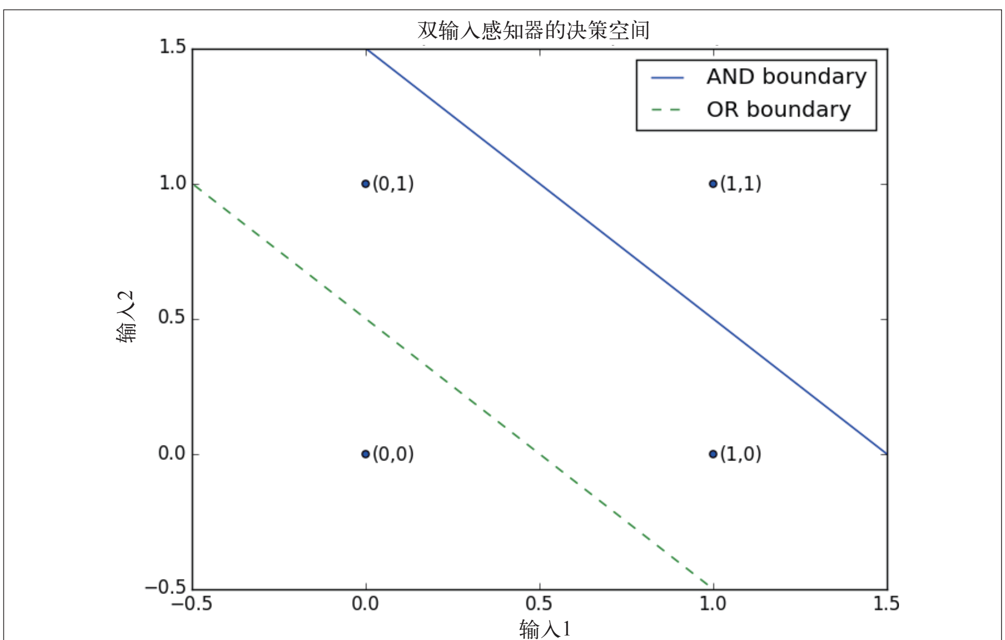


图 18-1：双输入感知器的决策空间

同样，我们还可以建立一个非门（即 NOT，它只有一个输入端，并且会把输入的 1 转换为 0，反之亦然），代码如下所示：

```
weights = [-2]
bias = 1
```

不过，有些问题是单个感知器所无法解决的，比如，无论你如何尝试，都无法通过一个感知器来构建异或门（XOR），即两个输入不同时输出为 1，否则输出为 0。这种情况下，我们就需要使用更加复杂的神经网络了。

当然，在建立逻辑门的时候，根本无需惟妙惟肖地模仿神经元，看一眼下面的代码你就明白了：

```
and_gate = min
or_gate = max
xor_gate = lambda x, y: 0 if x == y else 1
```

就像真实的神经元那样，当你将它们连接起来时，就会发生许多有趣的事情。

18.2 前馈神经网络

大脑的拓扑结构极为复杂，我们可以近似地把它看作一个理想化的前馈（feed-forward）神经网络，该网络由多层构成，每层由众多神经元组成，然后逐层相连。一般情况下，前馈神经网络会有一个输入层（接收输入信号，然后无需修改直接向前馈送），一个或者多个“隐藏层”（每层都是由神经元组成，这些神经元以前一层的输出作为其输入，进行某些计算，并将结果传递给下一层），以及一个输出层（这一层提供最终输出）。

正如感知器那样，每个（非输入）神经元的每个输入和偏移项都会有一个权重。为简单起见，我们将偏移项放到权重向量的末尾，并且所有神经元的偏移项的输入都是 1。

类似感知器那样，对于每个神经元而言，其输入与权重之积需要加总处理。不同之处在于，这里不是直接输出 step_function 函数应用于输入与权重之积的结果，而是将其平滑处理之后，输出一个近似值。准确地说，这里使用的是 sigmoid 函数，见图 18-2。

```
def sigmoid(t):
    return 1 / (1 + math.exp(-t))
```

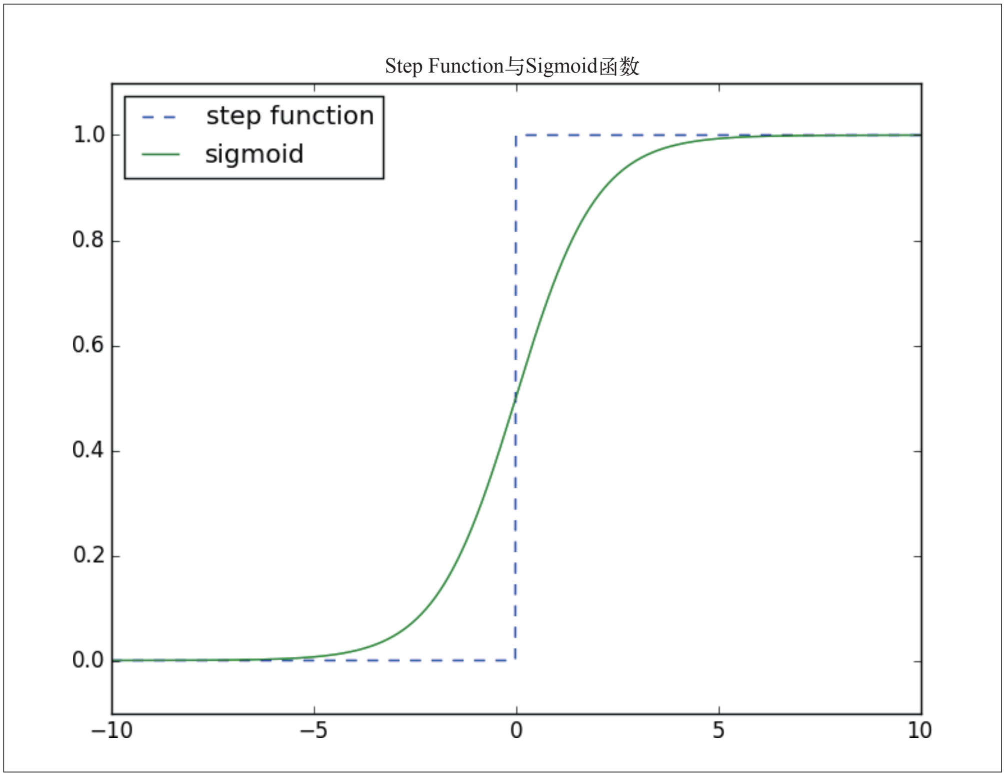


图 18-2：sigmoid 函数

为什么使用 `sigmoid` 函数，而不是更为简单的 `step_function` 函数呢？因为要训练神经网络，就得使用微积分，而要使用微积分，就得使用光滑函数。我们知道，阶梯函数无法确保处处连续，但是 `sigmoid` 函数却是它们一个非常好的平滑近似函数。

> 你可能还记得，我们在第 16 章也用过 `sigmoid` 函数，只不过当时我们称其为 `logistic` 函数。实际上，“sigmoid”指的是函数的外形，而“logistic”指的是这种特定的函数，但是人们经常将两者等价使用。

这样，我们就能计算其输出了，代码如下所示：

```
def neuron_output(weights, inputs):
    return sigmoid(dot(weights, inputs))
```

有了这个函数，我们就可以将神经元简单表示成一个权重列表，列表的长度等于神经元输入数量加 1，因为还要加上偏移项的权重。这样，神经网络就可以用各个（非输入）层组成的列表来表示，其中每一层就是该层内的神经元所组成的一个列表。

也就是说，神经网络可以用（权重）列表的（神经元）列表的（层）列表来表示。

有了这种表示方法，神经网络用起来就会非常简便：

```
def feed_forward(neural_network, input_vector):
    """takes in a neural network
    (represented as a list of lists of lists of weights)
    and returns the output from forward-propagating the input"""

    outputs = []

    # 每次处理一层
    for layer in neural_network:
        input_with_bias = input_vector + [1]              # 增加一个偏倚输入
        output = [neuron_output(neuron, input_with_bias)  # 计算输出
                  for neuron in layer]                    # 每一个神经元
        outputs.append(output)                            # 记住它

        # 然后下一层的输入就是这一层的输出
        input_vector = output

    return outputs
```

如今，我们无需使用感知器就能建立异或门了，这样事情就变得简单多了。所以，我们只需要调整权重，就能使得 neuron_outputs 非常接近 1 或 0 了：

```
xor_network = [# hidden layer
               [[20, 20, -30],      # 'and'神经元
                [20, 20, -10]],     # 'or'神经元
               # output layer
               [[-60, 60, -30]]]    # '第二次输入不同于第一次输入'神经元

for x in [0, 1]:
    for y in [0, 1]:
        # feed_forward生成每个神经元的输出
        # feed_forward[-1]是输出层神经元的输出
        print x, y, feed_forward(xor_network,[x, y])[-1]

# 0 0 [9.38314668300676e-14]
# 0 1 [0.9999999999999059]
# 1 0 [0.9999999999999059]
# 1 1 [9.383146683006828e-14]
```

借助于隐藏层，我们就能把一个“与”神经元和一个“或”神经元的输出馈送至“第一个输入不同于第二个输入”神经元了。这个网络所做的工作，就是判断“或运算的结果不同于与运算的结果”，这实际上就是在执行异或运算，见图 18-3。

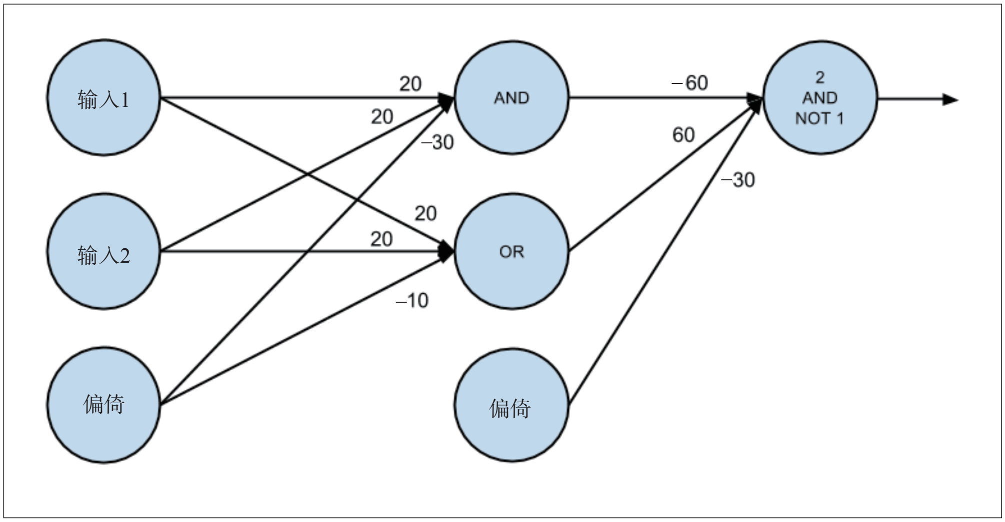


图 18-3：用于实现异或运算的神经网络

18.3 反向传播

通常情况下，我们是不会以手动方式建立神经网络的。部分原因在于，神经网络解决的是比较大型的问题，比如图象识别可能会用到数百或成千上万的神经元。还有一部分原因是，我们通常无法“通过推理得出”这些神经元的安排方式。

相反，我们会像往常一样使用数据用来训练神经网络。一个流行的训练算法是反向传播（backpropagation），它与前面介绍过的梯度下降法比较类似。

假如我们有一个训练集，其中含有输入向量和相应的目标输出向量。例如，前面 `xor_network` 例子中的输入向量为 `[1, 0]`，对应的目标输出端向量为 `[1]`。同时，假定我们的网络已经拥有一组权重，那么接下来，我们就需要使用以下算法来调整这些权重。

1. 在输入向量上运行 `feed_forward`，从而得到网络所有神经元的输出。
2. 这样，每个输出神经元都会得到一个误差，即目标值与输出值之差。
3. 计算作为神经元权重的函数的误差的梯度，然后根据误差降低最快的方向调整权重。
4. 将这些输出误差反向传播给隐藏层以便计算相应误差。
5. 计算这些误差的梯度，并利用同样的方式调整隐藏层的权重。

一般情况下，这个算法需要在整个训练集上多次迭代，直到网络收敛为止：

```
def backpropagate(network, input_vector, targets):

    hidden_outputs, outputs = feed_forward(network, input_vector)
```

```
# the output * (1 - output) is from the derivative of sigmoid
output_deltas = [output * (1 - output) * (output - target)
                 for output, target in zip(outputs, targets)]

# adjust weights for output layer, one neuron at a time
for i, output_neuron in enumerate(network[-1]):
    # focus on the ith output layer neuron
    for j, hidden_output in enumerate(hidden_outputs + [1]):
        # adjust the jth weight based on both
        # this neuron's delta and its jth input
        output_neuron[j] -= output_deltas[i] * hidden_output

# back-propagate errors to hidden layer
hidden_deltas = [hidden_output * (1 - hidden_output) *
                  dot(output_deltas, [n[i] for n in output_layer])
                 for i, hidden_output in enumerate(hidden_outputs)]

# adjust weights for hidden layer, one neuron at a time
for i, hidden_neuron in enumerate(network[0]):
    for j, input in enumerate(input_vector + [1]):
        hidden_neuron[j] -= hidden_deltas[i] * input
```

实际上，以上代码所做的事情无异于显式写出与权重有关的均方误差，并应用第 8 章建立的 `minimize_stochastic` 函数。

就本例而言，显式写出梯度函数非常麻烦。如果你熟悉微积分和链式法则，那么这些数学细节对你来说还算简单，但是直接用文字表述的话（“误差函数对神经元 i 至神经元 j 输入上的权重求偏导数”）则相当无趣。

18.4　实例：战胜CAPTCHA

为了确保在网站注册的是真实的人而非“机器人”，负责产品管理的副总想在注册过程中添加 CAPTCHA 功能。

准确地讲，他想向用户展示一个数字的图片，并要求他们输入数字，以此证明他们确实是真实的人。

你告诉他这难不倒计算机，但是他却不信，所以你打算写一个可以轻松搞定这个问题的程序来说服他。

下面，我们将通过 5 × 5 像素的图片来显示各个数字：

```
@@@@@  ..@..  @@@@@  @@@@@  @...@  @@@@@  @@@@@  @@@@@  @@@@@  @@@@@
@...@  ..@..  ....@  ....@  @...@  @....  @....  ....@  @...@  @...@
@...@  ..@..  @@@@@  @@@@@  @@@@@  @@@@@  @@@@@  ....@  @@@@@  @@@@@
@...@  ..@..  @....  ....@  ....@  ....@  @...@  ....@  @...@  ....@
@@@@@  ..@..  @@@@@  @@@@@  ....@  @@@@@  @@@@@  ....@  @@@@@  @@@@@
```

由于我们的神经网络是以数字组成的向量作为其输入的，所以我们将每个图像转换为长度为 25 的向量，其元素的值是 1（“这个像素位于该图像中”）或 0（“这个像素不在该图像中”）。

例如，数字 0 可以表示为：

```
zero_digit = [1,1,1,1,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,1,1,1,1]
```

我们希望神经网络给出的结果能够指向一个具体的阿拉伯数字，因此我们需要 10 种不同的输出结果。例如对于数字 4 来说，正确的输出结果将是：

```
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

那么，假如我们要按顺序输入 0 到 9，则相应的识别对象为：

```
targets = [[1 if i == j else 0 for i in range(10)]
           for j in range(10)]
```

因此，四号识别对象即 `targets[4]` 的正确输出结果为数字 4。

好了，现在可以建立我们的神经网络了：

```
random.seed(0)      # 得到重复的结果
input_size = 25     # 每个输入都是一个长度为25的向量
num_hidden = 5      # 隐藏层将含有5个神经元
output_size = 10    # 对于每个输入，我们需要10个输出结果

# 每一个隐藏神经元对每个输入都有一个权重和一个偏倚权重
hidden_layer = [[random.random() for __ in range(input_size + 1)]
                for __ in range(num_hidden)]

# 每一个输出神经元对每个隐藏神经元都有一个权重和一个偏倚权重
output_layer = [[random.random() for __ in range(num_hidden + 1)]
                for __ in range(output_size)]

# 神经网络是从随机权重开始的
network = [hidden_layer, output_layer]
```

这里，我们可以通过反向传播算法来训练我们的模型：

```
# 10 000次迭代看起来足够进行收敛
for __ in range(10000):
    for input_vector, target_vector in zip(inputs, targets):
        backpropagate(network, input_vector, target_vector)
```

它在训练集上效果很好：

```
def predict(input):
    return feed_forward(network, input)[-1]

predict(inputs[7])
# [0.026, 0.0, 0.0, 0.018, 0.001, 0.0, 0.0, 0.967, 0.0, 0.0]
```

这表明，输出数字 7 的神经元的值为 0.97，而其他输出神经元的值则非常小。

同时，我们还可以将其应用于不同的数字表示形式上，比如数字 3 可以用如下表示形式：

```
predict([0,1,1,1,0,  # .@@@.
         0,0,0,1,1,  # ...@@
         0,0,1,1,0,  # ..@@.
         0,0,0,1,1,  # ...@@
         0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.92, 0.0, 0.0, 0.0, 0.01, 0.0, 0.12]
```

我们的神经网络认为它看起来非常像 3，但是对于像如下这种形式表示的数字 8，输出结果中数字 5、8 和 9 的得分都不低：

```
predict([0,1,1,1,0,  # .@@@.
         1,0,0,1,1,  # @..@@
         0,1,1,1,0,  # .@@@.
         1,0,0,1,1,  # @..@@
         0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.0, 0.0, 0.55, 0.0, 0.0, 0.93, 1.0]
```

也许更大的训练集会有所帮助。

虽然神经网络的运行不是完全透明的，但我们可以通过检查隐藏层的权重来了解它们的识别情况。特别地，我们可以把每个神经元的权重绘制为 5×5 的网格，该网格是与 5×5 的输入相对应的。

现实中，你可能希望数值为 0 的权重的颜色为白色，大于 0 的权重的绝对值越大，颜色（比如说）越绿，小于 0 的权重的绝对值越大，颜色（比如说）越红。令人遗憾的是，这在黑白色的书中是无法做到的。

相反，我们会用白色表示值为 0 的权重，其值离 0 越远的权重颜色越暗。同时，利用阴影线来表示符号为负的权重。

为此，我们需要用到函数 pyplot.imshow——这是我们之前没提及的一个函数。利用它，我们可以逐像素地绘制图像。通常情况下，这个函数对于数据科学没有很大用途，但在这里，该函数意义非凡：

```
import matplotlib
weights = network[0][0]             # 隐藏层的第一个神经元
abs_weights = map(abs, weights)     # 阴影部分只取决于绝对值
```

```
grid = [abs_weights[row:(row+5)]      # 将权重转化为5x5的网格
        for row in range(0,25,5)]     # [weights[0:5], ..., weights[20:25]]

ax = plt.gca()                        # 为了使用影线,我们需要轴

ax.imshow(grid,                       # 这里与plt.imshow一样
          cmap=matplotlib.cm.binary,  # 使用白-黑色度
          interpolation='none')       # 不进行插值处理

def patch(x, y, hatch, color):
    """return a matplotlib 'patch' object with the specified
    location, crosshatch pattern, and color"""
    return matplotlib.patches.Rectangle((x - 0.5, y - 0.5), 1, 1,
                                        hatch=hatch, fill=False, color=color)

# 用交叉影线表示负权重
for i in range(5):                    # 行
    for j in range(5):                # 列
        if weights[5*i + j] < 0:      # row i, column j = weights[5*i + j]
            # 加上黑白影线,这样无论深浅就都可见了
            ax.add_patch(patch(j, i, '/',  "white"))
            ax.add_patch(patch(j, i, '\\', "black"))

plt.show()
```

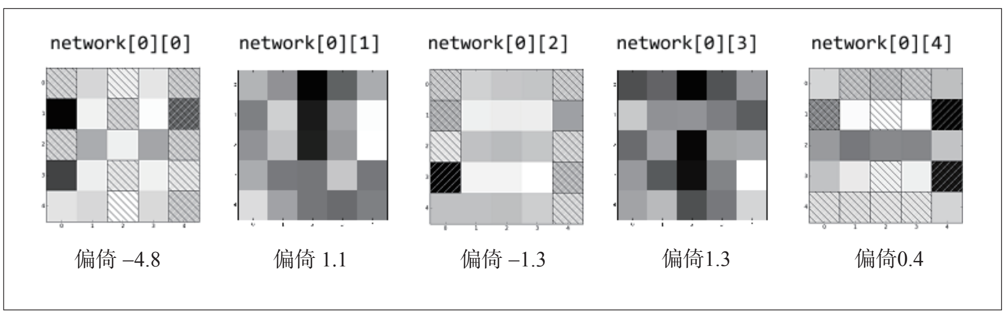


图 18-4：隐藏层的各个权重

通过图 18-4 可以看到，第一个隐藏神经元在左列和中间行的中心处的权重为正值，并且绝对值较大，而右列中的权重为负值，且绝对值较大。(此外，你还会发现它的偏倚项为负值，且绝对值较大，这意味着除非它“正在考察的”输入为正，否则很难被激活。)

事实上，对于这些输入来说，它的输出确实如我们所愿：

```
left_column_only = [1, 0, 0, 0, 0] * 5
print feed_forward(network, left_column_only)[0][0]  # 1.0

center_middle_row = [0, 0, 0, 0, 0] * 2 + [0, 1, 1, 1, 0] + [0, 0, 0, 0, 0] * 2
print feed_forward(network, center_middle_row)[0][0]  # 0.95

right_column_only = [0, 0, 0, 0, 1] * 5
print feed_forward(network, right_column_only)[0][0]  # 0.0
```

同样，中间的隐藏神经元似乎“喜欢”水平行而非两边的垂直行，且最后一个隐藏的神经元似乎“喜欢”中心行而非最右列。（其他两个神经元则很难解释。）

对我们以个性化形式表示的数字 3 运行这个神经网络，结果会是怎样呢？

```
my_three =  [0,1,1,1,0,  # .@@@.
             0,0,0,1,1,  # ...@@
             0,0,1,1,0,  # ..@@.
             0,0,0,1,1,  # ...@@
             0,1,1,1,0]  # .@@@.

hidden, output = feed_forward(network, my_three)
```

隐藏层的输出结果为：

```
0.121080  # 来自network[0][0],可能是受到了(1, 4)的影响
0.999979  # 来自network[0][1],(0, 2)和(2, 2)的贡献较大
0.999999  # 来自network[0][2],除(3, 4)之外皆为正值
0.999992  # 来自network[0][3],依旧是(0, 2)和(2, 2)的贡献较大
0.000000  # 来自network[0][4],除了中间一行外,其他皆为负值或零值
```

这将进入表示“3”的输出神经元中，相应权重为 network[-1][3]：

```
-11.61  # hidden[0]的权重
 -2.17  # hidden[1]的权重
  9.31  # hidden[2]的权重
 -1.38  # hidden[3]的权重
-11.47  # hidden[4]的权重
- 1.92  # 偏倚输入的权重
```

因此，这个神经元将计算：

```
sigmoid(.121 * -11.61 + 1 * -2.17 + 1 * 9.31 - 1.38 * 1 - 0 * 11.47 - 1.92)
```

正如我们所看到的，其值为 0.92。实际上，隐藏层是将 25 维空间计算成 5 个不同的分区，也就是说将每个 25 维的输入映射为 5 个数字。然后，每个输出神经元从这 5 个数字中挑出一个作为输出。

我们看到，my_three 落在分区 0（即只轻微激活了隐藏神经元 0）“下方”，前面是分区 1、2 和 3（即强烈激活那些隐藏神经元）“上部”，再往前些是分区 4 的底部（即所有神经元都没激活）。然后，10 个输出神经元中的每一个都会使用这 5 个激活单元来判断 my_three 是否为它们对应的数字。

18.5 延伸学习

- Coursera 提供了一门免费课程“机器学习的神经网络”（https://www.coursera.org/course/neuralnets）。在我编写本书时，最近的一次开课时间是 2012 年，不过相关课程材料仍然可用。

- Michael Nielsen 正在编写一本免费的书，书名为 *Neural Networks and Deep Learning*（http://neuralnetworksanddeeplearning.com/）。当你阅读本书时，很可能它已经写完了。
- PyBrain（http://pybrain.org/）是一个相当简单的 Python 神经网络库。
- Pylearn2（http://deeplearning.net/software/pylearn2/）是一个更加高级同时也更难使用的神经网络库。

第19章

聚类分析

使吾辈得以类聚者，热情而非疯狂也。

——罗伯特·赫里克

本书中的大多数算法都是所谓的监督学习方法，因为它们都是以一组标注过的数据作为起点的，并且在此基础上为新的、未标注过的数据做出预测。然而，本章介绍的聚类分析却是一种无监督学习方法，也就是说，它可以利用完全未经标注的数据（也可以使用标注过的数据，但我们忽略这些标签）进行工作。

19.1 原理

每当你观察某些数据源时，很可能会发现数据会以某种形式形成聚类（cluster）。例如，展示百万富翁居住地的数据集中，数据点很可能在贝弗利山和曼哈顿等地方形成聚类。而在展示人们每周工作时间（以小时为单位）的数据集中，数据则很可能聚集在 40 附近（并且，如果这些数据来自法律规定人们每周至少工作 20 个小时的国家的话，那么这些数据很可能就会聚集在 19 左右。）对于登记选民的人口统计的数据集，则可能形成多种集群（例如“足球妈妈”“无聊的退休人员”“待业千禧”等），这些群体正是民意调查和政治顾问所要密切关注的。

与之前见到的问题不同，这种问题通常没有“正确”的聚类。一个可选的聚类方案是将某些“待业千禧”与“大学毕业生”分为一伙，而将另一些“待业千禧”与“啃老族”分为一组。当然，很难说哪种方案肯定比其他方案要好，但是，对于每一种方案而言，都可以按照自己的“优良聚类”标准不断进行优化。

此外，这些聚类本身无法对自己进行标注。要想标注的话，你必须考察每个聚类中的底层数据。

19.2　模型

对于我们来说，每一个输入都是 *d* 维空间中的一个向量（跟以前一样，我们还是使用数字列表来表示向量）。我们的目标是识别由类似的输入所组成的聚类，（有时）还要找出每个聚类的代表值。

例如，每个输入可以是博客文章的标题（我们可以设法用数字向量来表示它），那么在这种情况下，我们的目标可能是对相似的文章进行聚类，也可能是了解用户都在写什么博客内容。或者，假设我们有一张包含数千种（红、绿、蓝）颜色的图片，但是我们需要一个 10 色版本来进行丝网印刷。这时，聚类分析不仅可以帮助我们选出 10 种颜色，并且还能将“色差”控制在最小的范围之内。

k-均值算法（*k*-means）是一种最简单的聚类分析方法，它通常需要首先选出聚类 *k* 的数目，然后把输入划分为集合 S_1，…，S_k，并使得每个数据到其所在聚类的均值（中心对象）的距离的平方之和最小化。由于将 *n* 个点分配到 *k* 个聚类的方法非常多，所以寻找一个最优聚类方法是一件非常困难的事情。一般情况下，为了找到一个好的聚类方法，我们可以借助于迭代算法。

1. 首先从 *d* 维空间中选出选择 *k* 个数据点作为初始聚类的均值（即中心）。
2. 计算每个数据点到这些聚类的均值（即聚类中心）的距离，然后把各个数据点分配给离它最近的那个聚类。
3. 如果所有数据点都不再被重新分配，那么就停止并保持现有聚类。
4. 如果仍有数据点被重新分配，则重新计算均值，并返回到第 2 步。

利用第 4 章中学过的 `vector_mean` 函数，可以轻松创建如下所示的类来完成上述工作：

```
class KMeans:
    """performs k-means clustering"""

    def __init__(self, k):
        self.k = k          # 聚类的数目
        self.means = None   # 聚类的均值

    def classify(self, input):
        """return the index of the cluster closest to the input"""
        return min(range(self.k),
                   key=lambda i: squared_distance(input, self.means[i]))

    def train(self, inputs):
        # 选择k个随机点作为初始的均值
        self.means = random.sample(inputs, self.k)
```

```
        assignments = None
        while True:
            # 查找新分配
            new_assignments = map(self.classify, inputs)

            # 如果所有数据点都不再被重新分配,那么就停止
            if assignments == new_assignments:
                return

            # 否则就重新分配
            assignments = new_assignments

            # 并基于新的分配计算新的均值
            for i in range(self.k):
                # 查找分配给聚类i的所有的点
                i_points = [p for p, a in zip(inputs, assignments) if a == i]

                # 确保i_points不是空的,因此除数不会是0
                if i_points:
                    self.means[i] = vector_mean(i_points)
```

下面让我们来看看其中的原理。

19.3　示例：聚会

为了庆祝 DataSciencester 的发展壮大，用户回馈部门的副总决定针对你家乡的用户组织几场私人聚会，并赞助啤酒、披萨和 DataSciencester T 恤。由于你了解所有当地用户的住址（如图 19-1），所以他想让你来选择聚会的地点，以方便大家的参与。

根据具体的观察方式，你可能会发现有两个或三个用户群。（这很容易看出来，因为这里的数据只有两个维度。但是随着维度的增加，对眼神的挑战就会越来越大。）

首先，我们假设他提供的预算足以组织三次聚会。然后你来到计算机前面输入下列代码：

```
random.seed(0)          # 因此你得到的结果与我的一样
clusterer = KMeans(3)
clusterer.train(inputs)
print clusterer.means
```

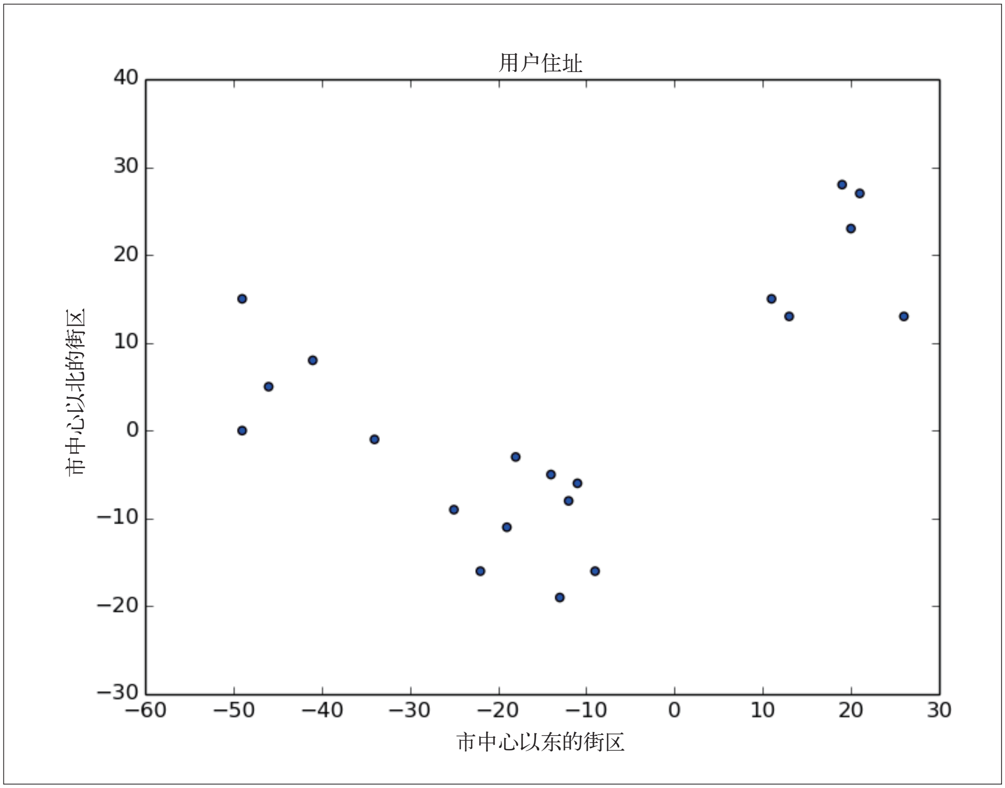


图 19-1：你家乡所在地的用户住址

你发现用户主要居住在以 [−44.5]、[−16，−10] 和 [18，20] 为中心的三个区域中，因此，你打算在这三个位置附近寻找聚会场地（见图 19-2）。

你将这些汇报给了副总，但是他却告诉你目前的预算仅够组织两次聚会了。

“没问题，”你说：

```
random.seed(0)
clusterer = KMeans(2)
clusterer.train(inputs)
print clusterer.means
```

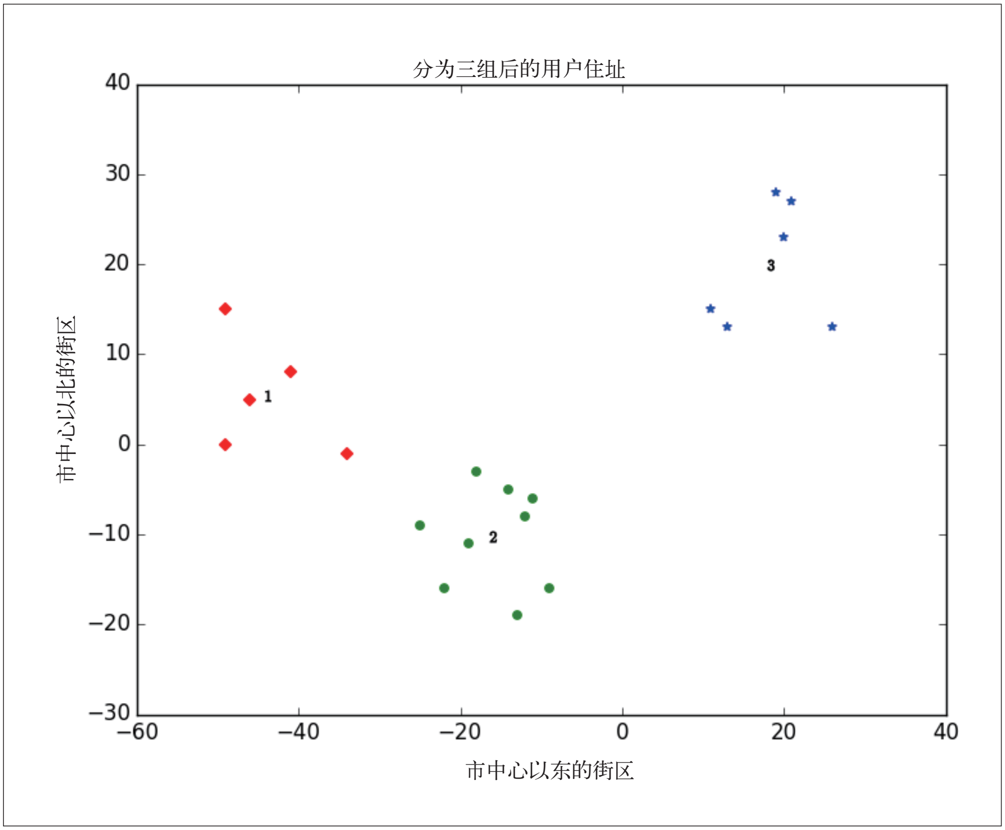


图 19-2：分为三组后的用户住址

如图 19-3 所示，某次聚会地点仍然定在 [18，20] 位置附近，而另一次聚会的地点则定在 [−26，−5] 位置附近。

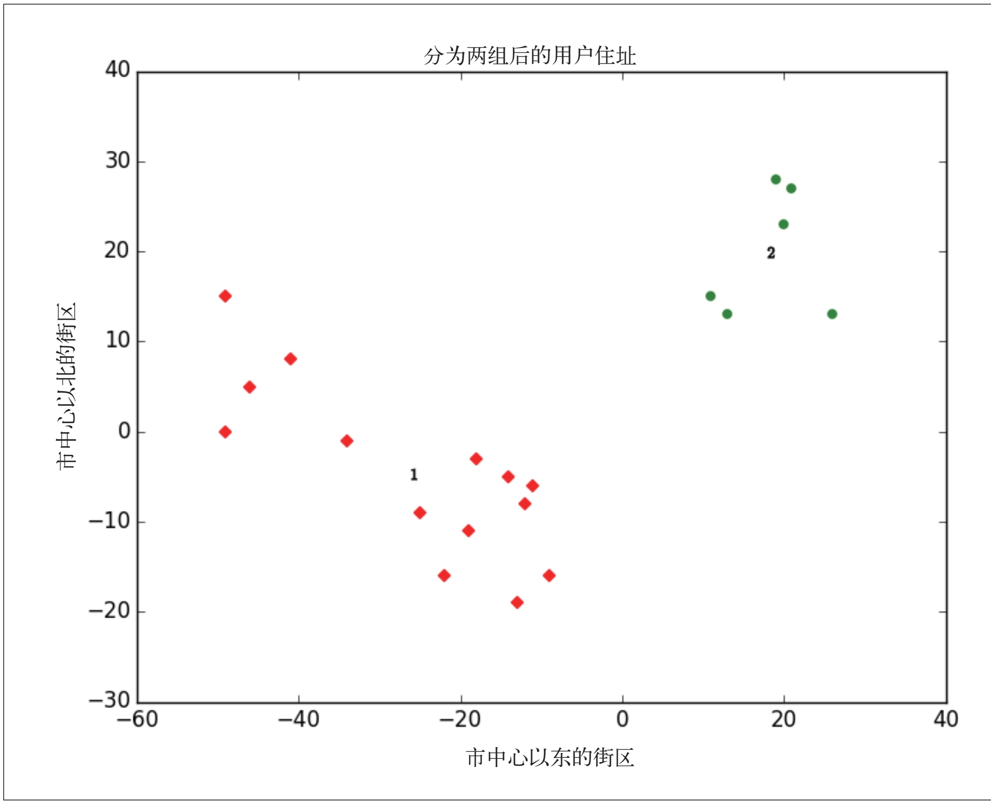


图 19-3：分为两组后的用户住址

19.4 选择聚类数目*k*

在前一个例子中，聚类数目 *k* 的选择是由外部因素决定的，我们无法控制。但是通常情况下，事情并非如此。*k* 的选择方法可谓五花八门，一个比较易于理解的方法是以误差（即每个数据点到所在聚类的中心的距离）的平方之和作为 *k* 的函数，画出该函数的图像，并在其“弯曲”的地方寻找合适的取值：

```
def squared_clustering_errors(inputs, k):
    """finds the total squared error from k-means clustering the inputs"""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = map(clusterer.classify, inputs)

    return sum(squared_distance(input, means[cluster])
               for input, cluster in zip(inputs, assignments))

# 现在画出1至len(输入)的聚类图
```

```
ks = range(1, len(inputs) + 1)
errors = [squared_clustering_errors(inputs, k) for k in ks]

plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("误差的平方之和")
plt.title("总误差与聚类数目")
plt.show()
```

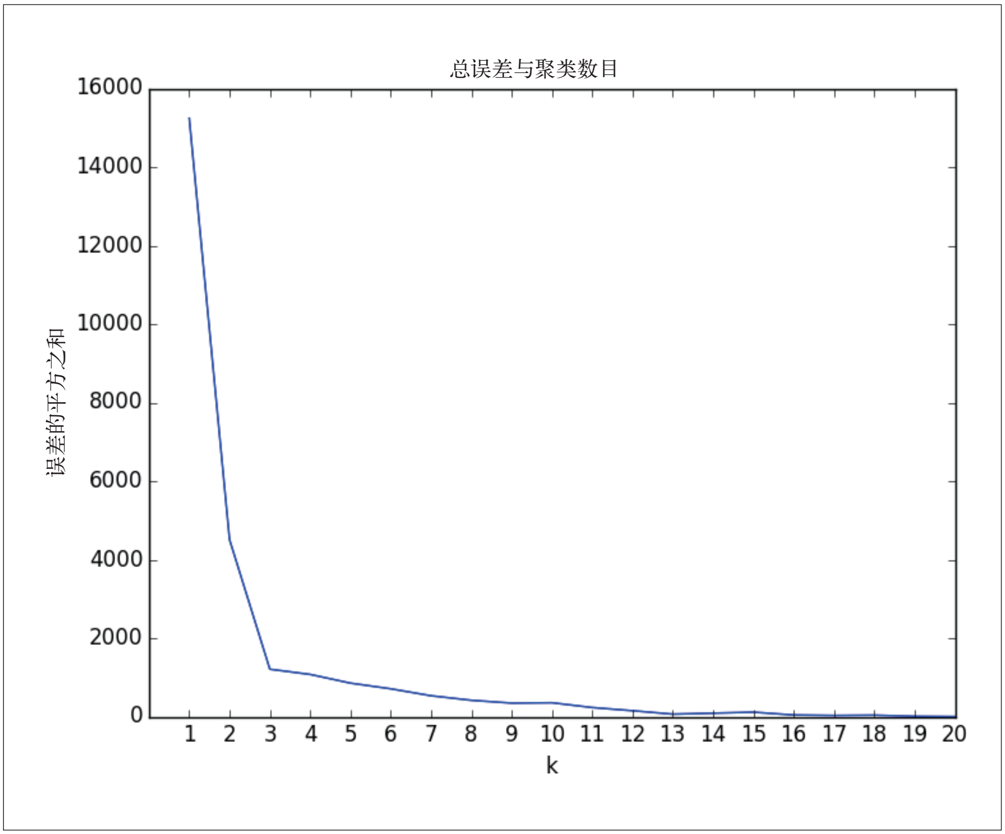


图 19-4：选择聚类数目 *k*

从图 19-4 可以看出，这种方法得到的结果与我们最初的目测值是相符的，也就是说 3 是一个“合适”的聚类数目。

19.5 示例：对色彩进行聚类

负责周边产品的副总设计了一款美观的 DataSciencester 便签，希望你能够在聚会上分发给用户。令人遗憾的是，你的便签打印机功能有限，每张便签上面最多只能打出五种色彩。

同时，由于负责美术的副总正在休假，因此，负责周边产品的副总向你咨询能否将其设计改为只包含五种颜色。

我们知道，计算机图像可以表示为像素的二维阵列，其中每个像素本身就是一个三维向量（red，green，blue），代表了该像素的颜色（https://en.wikipedia.org/wiki/RGB_color_model）。

为了得到图像的五色版本，我们需要执行下列步骤：

1. 选择五种颜色
2. 给每个像素从中挑选一种颜色

事实上，这个工作非常适合用 k-means 算法来做，因为该算法能够将像素划分为红－绿－蓝空间中的五个聚类。之后，我们只要将这些聚类中的像素用其中间色来重新着色就可以了。

首先，我们需要设法将图像加载到 Python 中。事实上，这可以借助 matplotlib 来实现：

```
path_to_png_file = r"C:\images\image.png"   # 不管你的图像在哪里
import matplotlib.image as mpimg
img = mpimg.imread(path_to_png_file)
```

实际上，img 在幕后是作为一个 NumPy 数组来处理的，不过就这里来说，我们可以将其视为以列表为元素的列表所组成的列表。

这里，img[i][j] 表示第 i 行第 j 列的像素，并且每个像素都由一个取值范围介于 0 和 1 之间的 [red, green, blue] 数字列表来指定其颜色：

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

特别是，我们可以将所有像素放到一个扁平化的列表中：

```
pixels = [pixel for row in img for pixel in row]
```

然后将其送入我们的聚类模型：

```
clusterer = KMeans(5)
clusterer.train(pixels)   # 这可能会花一些时间
```

一旦完成，我们得到了一张具有相同格式的新图像：

```
def recolor(pixel):
    cluster = clusterer.classify(pixel)        # 最近的聚类的索引
    return clusterer.means[cluster]            # 最近的聚类的均值

new_img = [[recolor(pixel) for pixel in row]   # 改变这一行像素的颜色
           for row in img]                     # 图像的每一行
```

接下来，我们就可以通过 plt.imshow() 来显示该图像了：

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

当然，在只有黑白色的书中无法展示这种色彩的效果，所以我们在图 19-5 中显示了一张全彩色图片的灰度图，以及一张采用这种技术将其降低到五种颜色后的灰度图：



图 19-5：原始图像以及利用 5-means 去色后的效果

19.6 自下而上的分层聚类

另一种聚类方法是采用自下而上的方式“培养”聚类，为此，我们可以借助下列方式：

1. 利用每个输入构成一个聚类，当然每个聚类只包含一个元素；
2. 只要还剩余多个聚类，就找出最接近的两个，并将它们合二为一 。

最后，我们将得到一个包含所有输入的巨大的聚类。如果我们将合并顺序记录下来，就可以通过拆分的方法来重建任意数量的聚类。举例来说，如果我们想得到 3 个聚类，那么只要撤销最后两次合并就可以了。

我们将使用一种非常简单的方法来表示聚类。首先，我们的数值将进入叶（leaf）聚类中，这时我们将其表示为一元组：

```
leaf1 = ([10, 20],)   # 要创建一元组,需要在末尾加一个逗号
leaf2 = ([30, -15],)  # 否则Python会把括号当成单纯的括号
```

我们通过合并上面的聚类来培育新的聚类，并将其记为二元组（合并次序，子聚类）：

```
merged = (1, [leaf1, leaf2])
```

我们稍后会介绍合并次序，但现在不妨先来创建一些辅助函数：

```
def is_leaf(cluster):
    """a cluster is a leaf if it has length 1"""
    return len(cluster) == 1

def get_children(cluster):
    """returns the two children of this cluster if it's a merged cluster;
    raises an exception if this is a leaf cluster"""
    if is_leaf(cluster):
        raise TypeError("a leaf cluster has no children")
    else:
        return cluster[1]

def get_values(cluster):
    """returns the value in this cluster (if it's a leaf cluster)
    or all the values in the leaf clusters below it (if it's not)"""
    if is_leaf(cluster):
        return cluster      # 已经是一个包含值的一元组
    else:
        return [value
                for child in get_children(cluster)
                for value in get_values(child)]
```

为了合并相距最近的聚类，我们需要明确聚类之间的距离的概念。为此，我们将使用两个聚类的元素之间的最小距离，据此将两个挨得最近的聚类合并（但有时会产生巨大的链式聚类，但是聚类之间却挨得不是很紧）。如果想得到紧凑的球状聚类，可使用最大距离，而不是最小距离，因为使用最大距离合并聚类时，它会尽力将两者塞进一个最小的球中。实际上，这两种距离都很常用，就像平均距离也很常用一样：

```
def cluster_distance(cluster1, cluster2, distance_agg=min):
    """compute all the pairwise distances between cluster1 and cluster2
    and apply _distance_agg_ to the resulting list"""
    return distance_agg([distance(input1, input2)
                         for input1 in get_values(cluster1)
                         for input2 in get_values(cluster2)])
```

我们将借助合并次序踪迹（slot）来跟踪合并的顺序。这个数字越小，表示合并的次序越靠后。这意味着，当我们想分拆聚类的时候，可以根据合并次序的值，从最小到最大依次进行。由于叶聚类不是合并而来的（这意味着无需分拆它们），因此，我们将它们合并次序的值规定为无穷大：

```
def get_merge_order(cluster):
    if is_leaf(cluster):
        return float('inf')
    else:
        return cluster[0]  # merge_order是二元组中的第一个元素
```

现在我们可以创建聚类算法了：

```
def bottom_up_cluster(inputs, distance_agg=min):
    # 最开始每个输入都是一个叶聚类/一元组
    clusters = [(input,) for input in inputs]

    # 只要剩余一个以上的聚类……
    while len(clusters) > 1:
        # 就找出最近的两个聚类
        c1, c2 = min([(cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]],
                      key=lambda (x, y): cluster_distance(x, y, distance_agg))

        # 从聚类列表中将它们移除
        clusters = [c for c in clusters if c != c1 and c != c2]

        # 使用merge_order = 剩余聚类的数目来合并它们
        merged_cluster = (len(clusters), [c1, c2])

        # 并添加它们的合并
        clusters.append(merged_cluster)

    # 当只剩一个聚类时,返回它
    return clusters[0]
```

它的使用方法非常简单：

```
base_cluster = bottom_up_cluster(inputs)
```

这将得到一个聚类，简单表示如下：

```
(0, [(1, [(3, [(14, [(18, [([19, 28],),
                           ([21, 27],)]),
                     ([20, 23],)]),
               ([26, 13],)]),
          (16, [([11, 15],),
                ([13, 13],)])]),
     (2, [(4, [(5, [(9, [(11, [([-49, 0],),
                               ([-46, 5],)]),
                         ([-41, 8],)]),
                    ([-49, 15],)]),
               ([-34, -1],)]),
          (6, [(7, [(8, [(10, [([-22, -16],),
                               ([-19, -11],)]),
                         ([-25, -9],)]),
                    (13, [(15, [(17, [([-11, -6],),
                                      ([-12, -8],)]),
                                ([-14, -5],)]),
                          ([-18, -3],)])]),
               (12, [([-13, -19],),
                     ([-9, -16],)])])])])
```

对于每一个合并而来的聚类，我都会将其子聚类纵向连接。当我们说“0 号聚类”为合并

次序为 0 的聚类时候，你可以将此理解为：

- 0 号聚类是由 1 号聚类和 2 号聚类合并得到的；
- 1 号聚类是由 3 号聚类和 16 号聚类合并得到的；
- 16 号聚类是由叶节点 [11，15] 和叶节点 [13，13] 合并得到的。
- 以此类推……

因为我们有 20 个输入，所以只要经过 19 次合并，便能得到这个聚类。第一个合并而来的聚类是通过合并叶节点 [19， 28] 和叶节点 [21， 27] 得到的。而最后一个合并而来的聚类则是 0 号聚类。

但是，一般情况下我们不喜欢这种繁琐的文字表达方法。（不过话又说回来，对于创建一个用户友好型的可视化聚类分层结构，这也算是一个有趣的练习。）下面让我们来写一个函数，使其可以通过执行适当次数的分拆动作来产生任意数量的聚类：

```
def generate_clusters(base_cluster, num_clusters):
    # 开始的列表只有基本聚类
    clusters = [base_cluster]

    # 只要我们还没有足够的聚类……
    while len(clusters) < num_clusters:
        # 选择上一个合并的聚类
        next_cluster = min(clusters, key=get_merge_order)
        # 将它从列表中移除
        clusters = [c for c in clusters if c != next_cluster]
        # 并将它的子聚累添加到列表中(即拆分它)
        clusters.extend(get_children(next_cluster))

    # 一旦我们有了足够的聚类……
    return clusters
```

举例来说，如果我们想要生成三个聚类，可以使用下列代码：

```
three_clusters = [get_values(cluster)
                  for cluster in generate_clusters(base_cluster, 3)]
```

利用下面的代码，我们可以轻松绘制其图形：

```
for i, cluster, marker, color in zip([1, 2, 3],
                                     three_clusters,
                                     ['D','o','*'],
                                     ['r','g','b']):
    xs, ys = zip(*cluster)  # 魔法般的解压方式
    plt.scatter(xs, ys, color=color, marker=marker)

    # 向聚类的均值添加一个数字
    x, y = vector_mean(cluster)
    plt.plot(x, y, marker='$' + str(i) + '$', color='black')
```

```
plt.title("利用最短距离得到的三个自下而上的聚类")
plt.xlabel("市中心以东的街区")
plt.ylabel("市中心以北的街区")
plt.show()
```

与 *k*- 均值算法相比，我们得到了一个大不相同的结果，具体如图 19-6 所示。

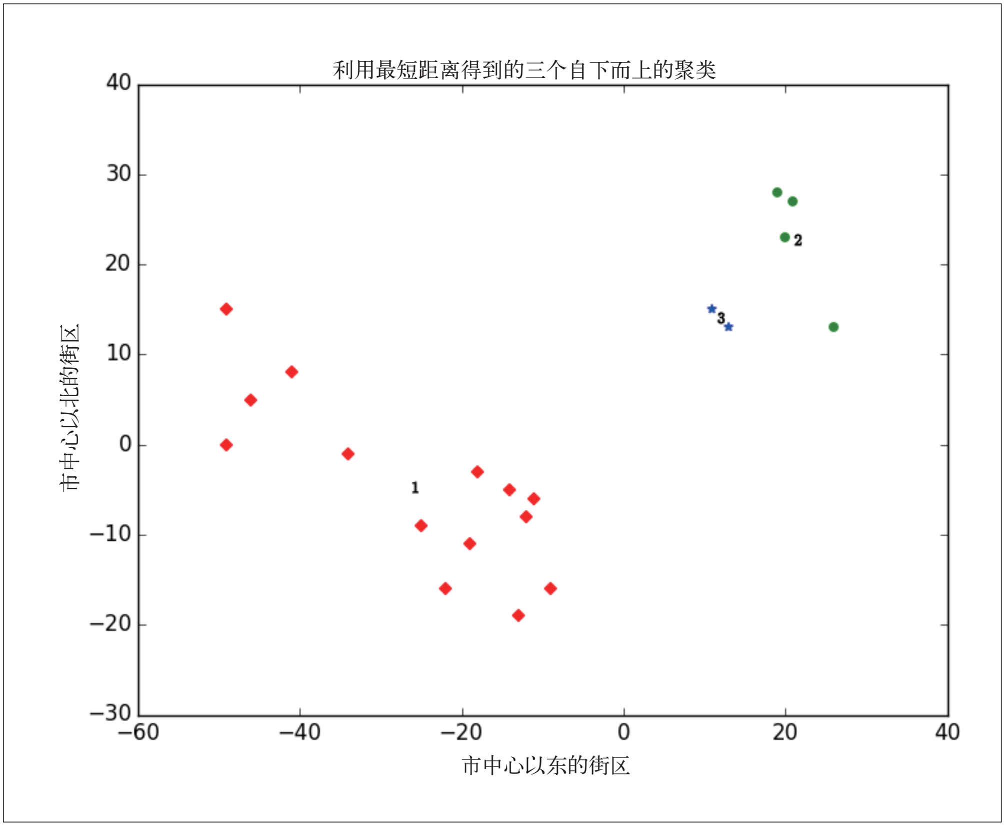


图 19-6：利用最短距离得到的三个自下而上的聚类

正如我们上面提到的，这是因为在 `cluster_distance` 函数中使用参数 `min` 时往往会得到链状聚类。相反，如果我们使用参数 `max`（这能得到更加紧凑的聚类），将得到看上去与图 19-7 所示的 3-means 无异的结果。

> 以上的 `bottom_up_clustering` 的实现代码相对来说已经很简单了，但是计算效率依然低得吓人。特别是，在每一步它都要重新计算每对输入之间的距离。更有效的实现方法是，预先算出每对输入之间的距离，然后在 cluster_distance 里面进行查找。一个真正高效的实现方法可能还需要存储上一步的 `cluster_distance`。

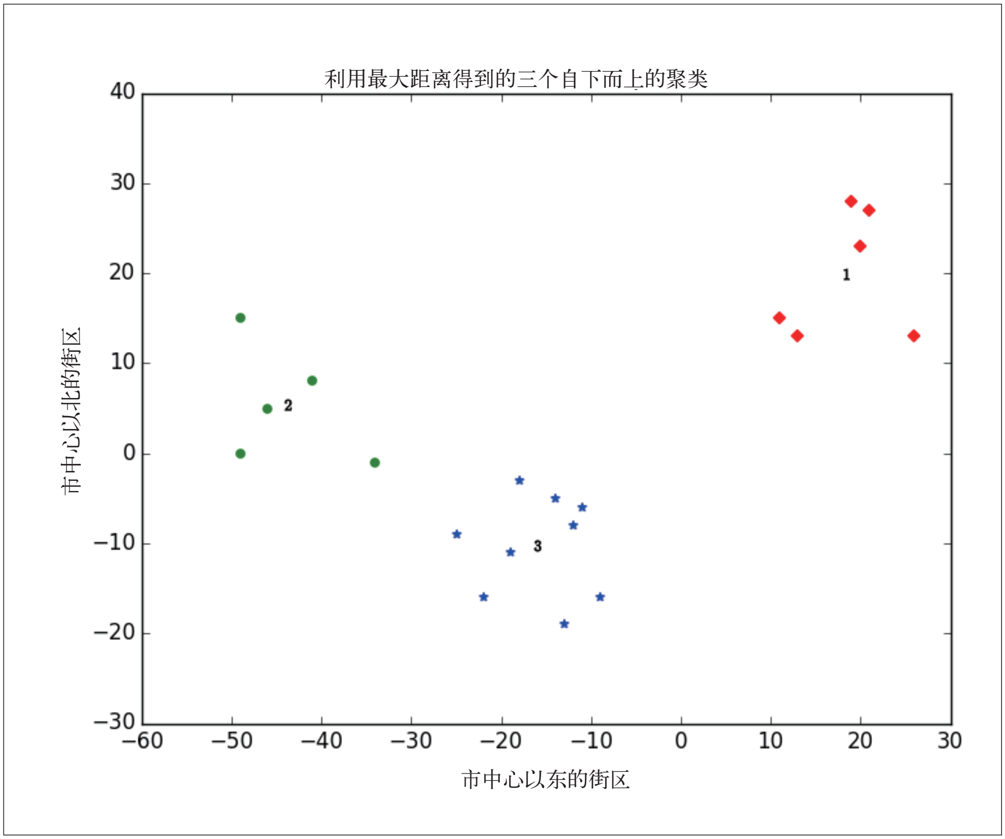


图 19-7：利用最大距离得到的三个自下而上的聚类

19.7 延伸学习

- `scikit-learn` 库中提供了一个单独的模块 `sklearn.cluster`（http://scikit-learn.org/stable/modules/clustering.html），其中含有多个聚类算法，包括 `KMeans` 和 `Ward` 分级聚类算法（该算法使用了不同的聚类合并规则）。
- Scipy（http://www.scipy.org/）模块也有两个聚类模型，即 `scipy.cluster.vq`（它使用 *k*-均值算法）模型和 `scipy.cluster.hierarchy`（它使用多种层次聚类算法）模型。

第20章

自然语言处理

他们刚从一场语言的盛宴上偷了些残羹冷炙回来。

——威廉·莎士比亚

自然语言处理（natural language processing，NLP）是指与语言有关的各种计算技术。这是一个广泛的领域，但我们这里只介绍几种相关的技术，它们简约却不简单。

20.1　词云

在第 1 章中，我们曾经计算过用户兴趣词汇的数量。为了使单词及其数量可视化，一种方法是使用词云，它不仅能够以艺术化的形式展示单词，而且还能使单词的大小与其数量呈正比。

但是，一般情况下，数据科学家并不看重词云，这在很大程度上是因为单词的布局没有任何特殊意义，顶多意味着“这里还有一些空，可以放上一个单词”而已。

当你不得不生成一个词云的时候，不妨考虑一下能否透过词的坐标传达某些东西。举例来说，假如你收集了一些与数据科学相关的流行语，那么对于每一个流行语，你可以用两个介于 0 至 100 之间的数字来描述，第一个数字代表它在招聘广告中出现的频次，第二个数字是在简历中出现的频次：

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
         ("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 60),
         ("data science", 60, 70), ("analytics", 90, 3),
         ("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0),
```

```
    ("actionable insights", 40, 30), ("think out of the box", 45, 10),
    ("self-starter", 30, 50), ("customer focus", 65, 15),
    ("thought leadership", 35, 35)]
```

词云的做法，只不过就是利用很酷的字体把各个单词布置到页面上罢了（图 20-1）。



图 20-1：由热门术语组成的词云

这看起来虽然整洁，但并没有告诉我们任何事情。一个更有趣的方法可能是将它们分散开来，利用水平位置表示其在招聘广告中的流行度，用垂直位置表示其在简历中的流行度，这样就能形象地传达一些信息（图 20-2）：

```
def text_size(total):
    """equals 8 if total is 0, 28 if total is 200"""
    return 8 + total / 200 * 20

for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
plt.xlabel("Popularity on Job Postings")
plt.ylabel("Popularity on Resumes")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

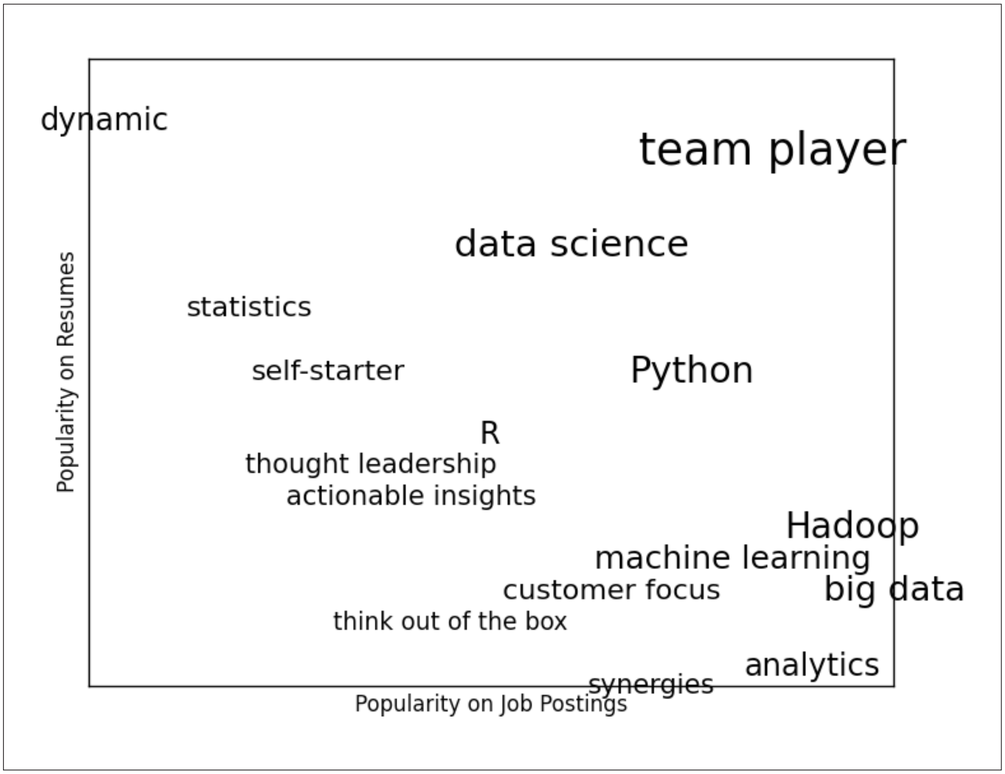


图 20-2：一个更有意义（尽管不如先前那么美观）的词云

20.2　n-grams模型

DataSciencester 负责搜索引擎营销的副总突发奇想，要创建数以千计的数据科学方面的 Web 页面，以便人们在搜索与数据科学有关的词语时，我们的网站能够在搜索结果中的排名更加靠前。（你试图向他解释，由于搜索引擎的算法已经足够聪明了，所以这种做法很难奏效，但是他根本就不听这一套。）

当然，他既不想亲自编写数以千计的网页，也不打算雇用一批“水军”来做这件事情。相反，他向你咨询是否可以通过编程方式来生成这些网页。为此，我们需要寻找某种方法来对语言进行建模。

这种方法当然是有的，比如首先搜集一批文档，然后利用统计方法得到一个语言模型。在我们的例子中，我们将从 Mike Loukides 的文章“什么是数据科学？”（https://www.oreilly.com/ideas/what-is-data-science）着手。

就像在第 9 章中所做的那样，我们将使用一些 Web 请求命令（`requests`）和 `BeautifulSoup` 来检索数据。不过，这里有几个问题需要引起我们的注意。

第一个问题是，文本中的单引号实际上就是 Unicode 字符 u"\u2019"。我们可以创建一个辅助函数，用正常的单引号来取代它们：

```
def fix_unicode(text):
    return text.replace(u"\u2019", "'")
```

第二个问题是，在获得了网页的文本之后，我们需要把它做成一个由单词和句号组成的序列（这样我们就可以知道句子的结尾在哪里）。为此，我们可以借助于 re.findall() 函数来完成：

```
from bs4 import BeautifulSoup
import requests
url = "http://radar.oreilly.com/2010/06/what-is-data-science.html"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')

content = soup.find("div", "entry-content")   # 找到entry-content div
regex = r"[\w']+|[\.]"                        # 匹配一个单词或一个句点

document = []

for paragraph in content("p"):
    words = re.findall(regex, fix_unicode(paragraph.text))
    document.extend(words)
```

当然，我们可以（也应该）进一步清理这些数据。文档中依然存在一些多余的文字（例如，第一个字“Section”就多余），同时，我们是利用句点来断句的（例如，遇到“Web 2.0”就会出问题）。此外，文档中还散布了一些标题和列表。尽管如此，这个文档已经可以凑合着用了。

将文本做成了单词序列之后，我们就可以通过以下方式对语言进行建模了：给定某个起始单词（比如“book”），我们可以找出源文档中所有在它后面出现过的那些单词（这里是“isn't”“a”“shows”“demonstrates”和“teaches”）。我们从中随机选择一个来作为下一个单词，然后重复这个过程，直到我们遇到一个句点为止，因为句点就意味着句子的结束。我们将这个模型称之为*二元模型*（bigram model），因为这完全是由原始数据中 2 个词（一个词对）同时出现的频率决定的。

那么起始单词呢？实际上，我们只要从句点*后面*的单词中随机选择就行了。首先，让我们预先计算出可能的单词语次转变。回想一下，对于 zip 来说，只要输入中有一个已经处理完毕，它就会停下来，因此，我们可以利用 zip(document, document[1:]) 求出文档中有多少对连续元素：

```
bigrams = zip(document, document[1:])
transitions = defaultdict(list)
for prev, current in bigrams:
    transitions[prev].append(current)
```

下面我们就可以生成句子了：

```
def generate_using_bigrams():
    current = "."   # 这意味着下一个单词是一个新句子的开头
    result = []
    while True:
        next_word_candidates = transitions[current]    # 双连词 (current, _)
        current = random.choice(next_word_candidates)  # 随机选择一个
        result.append(current)                         # 将其附加到结果中
        if current == ".": return " ".join(result)     # 如果是".",就完成了
```

它产生的那些句子都是些无意义的数据，不过你可以把这些句子放到网站上，以让网站看起来更能与数据科学挂钩。举例来说：

> If you may know which are you want to data sort the data feeds web friend someone on trending topics as the data in Hadoop is the data science requires a book demonstrates why visualizations are but we do massive correlations across many commercial disk drives in Python language and creates more tractable form making connections then use and uses it to solve a data.
>
> ——Bigram Model

如果我们使用三元模型（trigrams）的话，就能够降低这些句子无意义的程度。所谓三元模型，就是使用三个连续的词得到的模型。（更一般地讲，你还可以考虑由 *n* 个连续的单词得到的 *n*-grams 模型，不过对于我们来说，由三个词组成的就足够了。）现在，这种语次转变将取决于前两个单词：

```
trigrams = zip(document, document[1:], document[2:])
trigram_transitions = defaultdict(list)
starts = []

for prev, current, next in trigrams:

    if prev == ".":              # 如果前一个"单词"是个句点
        starts.append(current)   # 那么这就是一个起始单词

    trigram_transitions[(prev, current)].append(next)
```

需要注意的是，现在我们必须将这些起始词单独记录下来。我们可以使用几乎相同的方法来生成句子：

```
def generate_using_trigrams():
    current = random.choice(starts)   # 随机选择一个起始单词
    prev = "."                        # 前面加一个句点'.'
    result = [current]
    while True:
        next_word_candidates = trigram_transitions[(prev, current)]
        next_word = random.choice(next_word_candidates)
```

```
        prev, current = current, next_word
        result.append(current)

        if current == ".":
            return " ".join(result)
```

这次得到的句子看起来要好一些：

> In hindsight MapReduce seems like an epidemic and if so does that give us new insights into how economies work That’s not a question we could even have asked a few years there has been instrumented.
>
> ——Trigram Model

当然，它们之所以看起来更好一些，是因为生成过程的每一步中，所面临的选择要更少一些，甚至有时候只有一种选择。这就意味着生成的句子（或至少是长短语）经常跟原始数据中的一字不差。更多的数据会有所帮助；此外，如果从多篇数据科学方面的文章中收集 *n*-grams，收到的成效会更好。

20.3 语法

还有一种语言建模方法，那就是利用*语法规则*（grammar）来生成符合要求的句子。在小学的时候，我们就已经知道了词的词类及其组合方式。例如，如果你有一个非常糟糕的英语老师，你必定会认为句子都是由*名词*后面跟*动词*构成的。这样的话，如果给你一个由名词和动词组成的列表，你就可以根据这种规则来造句了。

下面，我们将定义一个稍微复杂的语法：

```
grammar = {
    "_S"  : ["_NP _VP"],
    "_NP" : ["_N",
             "_A _NP _P _A _N"],
    "_VP" : ["_V",
             "_V _NP"],
    "_N"  : ["data science", "Python", "regression"],
    "_A"  : ["big", "linear", "logistic"],
    "_P"  : ["about", "near"],
    "_V"  : ["learns", "trains", "tests", "is"]
}
```

我们约定，以下划线开头的名称表示语法*规则*，它们需要进一步展开；而其他名称是不需要进一步处理的*终端符号*。

例如，"_S" 是“句子”规则，其产生一个 "_NP"（“名词短语”）规则，后面紧跟一个 "_VP"（“动词短语”）规则。

动词短语规则可能会产生一个 "_V"（“动词”）规则，也可能会产生一个动词规则继之以名词短语规则。

请注意，"_NP" 规则所生成的规则中也包括其自身。我们知道，语法是可以递归的，因此，尽管这里这些语法非常有限，但是照样能够产生无穷多不同的句子。

那么，我们如何通过这些语法来生成句子呢？ 我们不妨从一个包含句子规则的列表 ["_S"] 着手。然后，我们将不断展开每一项规则，即从该规则的产物中随机选择一个来代替它。当我们的列表元素全部变成终端符号时，我们就可以停下来了。

举例来说，上述过程可能像下面这样：

```
['_S']
['_NP','_VP']
['_N','_VP']
['Python','_VP']
['Python','_V','_NP']
['Python','trains','_NP']
['Python','trains','_A','_NP','_P','_A','_N']
['Python','trains','logistic','_NP','_P','_A','_N']
['Python','trains','logistic','_N','_P','_A','_N']
['Python','trains','logistic','data science','_P','_A','_N']
['Python','trains','logistic','data science','about','_A', '_N']
['Python','trains','logistic','data science','about','logistic','_N']
['Python','trains','logistic','data science','about','logistic','Python']
```

我们如何实现它呢？ 首先，我们需要创建一个简单的辅助函数来识别终端符号：

```
def is_terminal(token):
    return token[0] != "_"
```

接下来，我们需要编写一个函数，将一个标记列表变成一个句子。首先，我们需要找到第一个非终结符号标记。如果我们找不到这种标记，那就意味着我们已经有一个完整的句子，可以收工了。

如果我们真的找到了一个非终端符号，那么就在其产物中随机选择一个。如果选中的是个终端符号（即一个单词），那么直接用它替换相应的标记即可。除此之外，如果选中的是一个由空格符分隔的非终端符标记，那么则需要进行拆分，并将其拼接到当前标记中。总之，我们的工作就是在一组新标记上不断重复这个过程。

上述过程可以通过下列代码实现：

```
def expand(grammar, tokens):
    for i, token in enumerate(tokens):

        # 跳过终端符号
        if is_terminal(token): continue
```

```
        # 如果这一步我们发现了一个非终端符号
        # 需要随机选择一个替代者
        replacement = random.choice(grammar[token])

        if is_terminal(replacement):
            tokens[i] = replacement
        else:
            tokens = tokens[:i] + replacement.split() + tokens[(i+1):]

        # 现在展开新的符号列表
        return expand(grammar, tokens)

    # 如果到达这一步,就找出了所有的终端符号,可以收工了
    return tokens
```

现在我们可以生成句子了：

```
def generate_sentence(grammar):
    return expand(grammar, ["_S"])
```

只要我们不断改变语法——例如添加更多的单词、添加更多的规则、添加各种词类等——就能得到足够多的网页来满足公司的需要。

实际上，当语法用于另一个方向时，会变得更加有趣。给定一个句子，我们就可以用语法来*解析*句子。这就能帮助我们识别主语和动词，从而理解句子的含义。

用数据科学来生成文本的确是个妙招，但更神奇的是，它还可以用来*理解*文本。（这方面的程序库请参阅 16.6 节“延伸学习”部分。）

20.4　题外话：吉布斯采样

根据一些概率分布来生成样本是非常简单的事情。我们可以通过以下这行代码得到一些均匀分布的随机变量：

```
random.random()
```

以及用以下代码得到一些正态分布的随机变量：

```
inverse_normal_cdf(random.random())
```

但某些概率分布却很难进行采样。当我们只知道一些条件分布时，可以通过吉布斯采样技术根据多维分布来生成样本。

例如，假设我们在掷两只骰子。这里用 x 表示第一骰子的点数，y 表示两个骰子的点数之和。假设我们要产生大量形如 (x, y) 的数据对，这种情况下，我们可以直接通过下列代码来轻松生成所需样本：

```
def roll_a_die():
    return random.choice([1,2,3,4,5,6])

def direct_sample():
    d1 = roll_a_die()
    d2 = roll_a_die()
    return d1, d1 + d2
```

但是，这里假设你只知道条件分布。在已知道 x 的条件下求 y 的分布是很容易的：如果你知道了 x 的值，那么 y 就有同等机会等于 $x+1$，$x+2$，$x+3$，$x+4$，$x+5$，$x+6$：

```
def random_y_given_x(x):
    """equally likely to be x + 1, x + 2, ... , x + 6"""
    return x + roll_a_die()
```

如果将已知条件反过来，事情会变得更加复杂。举例来说，如果你知道 y 为 2，那么 x 必定为 1（因为只有当两个骰子的点数都为 1 的时候，点数之和才可能为 2）。如果你知道 y 为 3，那么 x 有等同的机会为 1 或 2。类似地，如果 y 为 11，那么 x 要么为 5，要么为 6：

```
def random_x_given_y(y):
    if y <= 7:
        # 如果点数为7或以下的数,那么第一个骰子的点数有等同的机会为
        # 1, 2, …, (总点数 - 1)
        return random.randrange(1, y)
    else:
        # 如果点数为7或以上的数,那么第一个骰子的点数有等同的机会为
        # (total - 6), (总点数 - 5), …, 6
        return random.randrange(y - 6, 7)
```

吉布斯抽样的方法是先从任意（有效）的 x 和 y 值入手，然后不断用 y 条件下随机选择的 x 值替换原来的 x，并用 x 条件下随机选择的 y 值替换原来的 y。重复一定次数后，得到的 x 值和 y 值就可以作为根据无条件的联合分布获取的样本了：

```
def gibbs_sample(num_iters=100):
    x, y = 1, 2 # doesn't really matter
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

通过下列代码你会发现，这种取样方法与直接取样的效果相似：

```
def compare_distributions(num_samples=1000):
    counts = defaultdict(lambda: [0, 0])
    for _ in range(num_samples):
        counts[gibbs_sample()][0] += 1
        counts[direct_sample()][1] += 1
    return counts
```

我们将在接下来的部分使用这种取样方法。

20.5 主题建模

在第 1 章我们建立“你应该知道的数据科学家”推荐系统的时候，我们只是根据科学家与读者的兴趣是否严格匹配来进行推荐的。

要想理解我们的用户，更高级的方法是识别这些兴趣背后的相关主题。有一种叫作隐含狄利克雷分析（latent dirichlet analysis，LDA）的技术，常用来确定一组文档的共同主题。我们会用它来分析包含每个用户的兴趣的那些文档。

这里的 LDA 与第 13 章中的朴素贝叶斯分类器有一些相似之处，它们都是用于处理文档的概率模型。我们将尽量避开一些数学细节问题，但对于该模型的某些假设却不得不说，如下所述。

- 存在固定数目的主题，即 K 个。
- 有一个给每个主题在单词上的概率分布赋值的随机变量。你可以把这个分布看作是单词 w 在给定主题 k 中出现的概率。
- 还有另一个随机变量来指出每个文档在主题下面的概率分布。你可以将这个分布看作是文档 d 中各主题所占比重。
- 文档中各个单词的生成方式为，首先（根据文档的主题分布情况）随机选择一个主题，然后（根据该主题下面各单词的分布情况）随机选择一个单词。

特别地，我们要建立一个文档（`documents`）集合，其中每个文档都是一个单词的列表。

同时，我们还要建立一个相应的 `document_topics` 集合，以便给每个文档中的每个单词都指定一个主题（这里用 0 到 K–1 之间的一个数字表示）。

这样的话，第 4 个文档中的第 5 个单词就可以表示为：

```
documents[3][4]
```

而这个选定的单词对应的主题可以表示为：

```
document_topics[3][4]
```

这非常显式地定义了每个文档在各个主题上的分布情况，同时也隐式地定义了每个主题在各个单词上面的分布情况。

我们可以估算出主题 1 产生一个特定单词的可能性，方法是将主题 1 产生该单词的次数除以主题 1 产生任意单词的次数。（类似地，我们在第 13 章中建立垃圾邮件过滤器时，也曾经用每个单词出现在垃圾邮件中的次数与这些单词出现在垃圾邮件中的总次数进行过相应的比较。）

这些主题都只是些数字，不过，我们可以用其权重最大的单词给它们取一个描述性的名

称。接下来，我们只需设法生成 document_topics 即可。这时，吉布斯取样技术就派上用场了。

首先，我们以完全随机的方式给所有文档中的每个单词都赋予一个主题。现在，我们就以每次一个单词的方式遍历所有文档。对于给定的单词和文档，我们需要根据该文档中主题（当前）的分布情况以及相对于该主题各单词（当前）的分布情况来建立相应的权重。然后，我们会使用这些权重给这个单词选取新主题。如果将该过程迭代多次，我们就能利用主题 – 单词分布和文档 – 主题分布完成联合取样。

首先，我们需要一个函数，根据任意一组权重来随机选择一个索引：

```
def sample_from(weights):
    """returns i with probability weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random()      # 在0和总数之间均匀分布
    for i, w in enumerate(weights):
        rnd -= w                       # 返回最小的i
        if rnd <= 0: return i          # 因此weights[0] + … + weights[i] >= rnd
```

例如，如果你给它的权重为 [1, 3, 1]，那么五分之一的时间将返回 0，五分之一的时间将返回 1，同时还有五分之三的时间将返回 2。

我们的文档包含的是用户的各种兴趣，看起来可能像下面这样：

```
documents = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

我们将设法找出 4 个主题，即 K = 4。

为了计算抽样权重，我们需要明确几项计数。下面，我们先给它们创建相应的数据结构。

我们要统计每个文档中每个主题出现的次数，代码如下：

```
# 计数的一个列表,每个文档各有一个列表
document_topic_counts = [Counter() for _ in documents]
```

我们要统计每个主题中每个单词出现的次数，代码如下：

```
# 计数的一个列表,每个主题各有一个列表
topic_word_counts = [Counter() for _ in range(K)]
```

我们要知道每个主题中单词的总数，代码如下：

```
# 数字的一个列表，每个主题各有一个列表
topic_counts = [0 for _ in range(K)]
```

下面的代码统计每个文档中的单词总数：

```
# 数字的一个列表,每个文档各有一个列表
document_lengths = map(len, documents)
```

下面代码统计不同单词的数量：

```
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)
```

另外，下列代码可以用来统计文档的数量：

```
D = len(documents)
```

一旦掌握了这些数据，我们就可以了解（比如说）documents[3] 中与主题 1 相关的单词的数量，具体代码如下所示：

```
document_topic_counts[3][1]
```

同时，我们还可以找出与主题 2 相关的单词 nlp 出现的次数，具体代码如下所示：

```
topic_word_counts[2]["nlp"]
```

现在，我们已经为定义条件概率函数做好了准备。就像在第 13 章中那样，这里的每个主题和单词都需要有一个平滑项，来确保每个主题在任何文档中被选中的几率都不能为 0，同时保证每个单词在任何主题中被选中的几率也都不能为 0：

```
def p_topic_given_document(topic, d, alpha=0.1):
    """the fraction of words in document _d_
    that are assigned to _topic_ (plus some smoothing)"""

    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word, topic, beta=0.1):
    """the fraction of words assigned to _topic_
    that equal _word_ (plus some smoothing)"""
```

```
        return ((topic_word_counts[topic][word] + beta) /
                (topic_counts[topic] + W * beta))
```

然后利用下列代码给更新中的主题确定权重：

```
def topic_weight(d, word, k):
    """given a document and a word in that document,
    return the weight for the kth topic"""
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)

def choose_new_topic(d, word):
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

上面的 topic_weight 之所以如此定义，背后是有坚实的数学理论作为依据的，不过其中的数学细节已经超出了本书的讨论范围。不过从直观的角度来看，如果已知一个单词和所在文档，那么该单词属于某主题的概率取决于两个方面，即该主题属于该文档的可能性以及该单词属于该主题的可能性。

这就是我们所需要的全部零部件。下面，我们开始将每个单词随机指派给一个话题，并计入相应的计数器：

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]

for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

我们的目标是利用主题 - 单词分布和文档 - 主题分布进行联合采样。为此，我们可以通过之前定义的条件概率进行吉布斯采样，具体代码如下所示：

```
for iter in range(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):

            # 从计数中移除这个单词/主题
            # 以便它不会影响权重
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1

            # 基于权重选择一个新的主题
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic
```

```
            # 现在把它重新加到计数中
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

这些主题都是什么呢？它们只是些数字而已：0、1、2 和 3。如果我们要让它们拥有名称的话，必须亲自给它们取名。下面，让我们来找出权重较大的 5 个单词（见表 20-1），具体代码如下所示：

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0: print k, word, count
```

表20-1：每个主题中最常见的单词

主题0	主题1	主题2	主题3
Java	R	HBase	regression
Big Data	statistics	Postgres	libsvm
Hadoop	Python	MongoDB	scikit-learn
deep learning	probability	Cassandra	machine learning
artificial intelligence	Pandas	NoSQL	neural networks

根据这些数据，我们就可以给主题取名了：

```
topic_names = ["Big Data and programming languages",
               "Python and statistics",
               "databases",
               "machine learning"]
```

至此我们就清楚模型是如何将主题分配到每个用户的兴趣上面了：

```
for document, topic_counts in zip(documents, document_topic_counts):
    print document
    for topic, count in topic_counts.most_common():
        if count > 0:
            print topic_names[topic], count,
    print
```

其输出结果如下所示：

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4 databases 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5 machine learning 1
```

如此等等。假使我们被要求用“ands”作为某个主题的名称，那么我们很可能需要使用更

多的主题，尽管更可能的情况是我们没有足够的数据来学习。

20.6 延伸学习

- Natural Language Toolkit（http://www.nltk.org/）是 Python 语言一个流行（和非常全面）的 NLP 工具库。对于这个库，已经有专门的书籍（http://www.nltk.org/book/）对它进行全面的介绍，并且该书可以在线阅读。
- gensim（http://radimrehurek.com/gensim/）是一个用于主题建模的 Python 库，与从头开始建模相比，使用它来建模是一种更靠谱的途径。

第21章

网络分析

你跟周遭事物的所有联系定义了你是谁。

——亚伦·奥康奈尔

当我们面对许多有趣的数据问题时，如果把它们看作由各种类型的节点（node）和连接它们的边（edge）所构成的网络，那就再合适不过了。

例如，你的 Facebook 好友可以看作一个网络中的节点，而连接节点的边可以看作朋友关系。另外一个不太明显的例子是万维网本身，其中的每一个网页都是一个网络节点，而从一个页面到另一个页面的超链接则是这个网络的边。

Facebook 中的朋友关系都是相互的，也就是说，如果我是你的 Facebook 好友，那么你也肯定是我的好友。这样的话，我们就可以说这种网络中的边是*无方向的*（undirected）。但是，超链接却并非如此，即如果我的网站链接到白宫的网站，并不表示白宫的网站也会链接到我的网站。我们称这种类型的边为*有方向的*（directed）。

我们下面会介绍这两种类型的网络。

21.1　中介中心度

在第 1 章中，我们只能通过每位用户的好友数量来计算 DataSciencester 网络中的关键联系人。现在，我们已经有足够多的手段来寻找其他的计算方法了。下面，我们先来看看网络中的用户，具体如图 21-1 所示：

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

以及用户之间的好友关系：

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

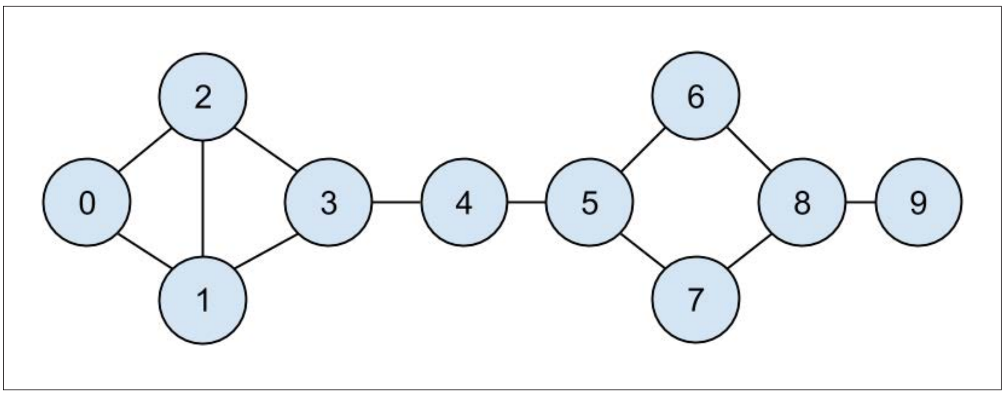


图 21-1：DataSciencester 网络

此外，我们还给每个用户的 dict 结构添加了相应的朋友列表：

```
for user in users:
    user["friends"] = []

for i, j in friendships:
    # 这能奏效是因为users[i]是id为i的用户
    users[i]["friends"].append(users[j]) # 添加i作为j的朋友
    users[j]["friends"].append(users[i]) # 添加j作为i的朋友
```

当时，我们对*度中心性*（degree centrality）的概念不甚满意，因为它与网络中直观展现在我们面前的关键联系人不甚相符。

另一种度量指标是*中介中心度*（betweenness centrality），它可以用来找出经常位于其他节点对之间的最短路径中的人。具体而言，中介中心度可以通过累加节点 j 和节点 k 之间经过节点 i 的最短路径所占比例，以及节点 j 和 k 之外所有的节点对中相应的比例来求出。

也就是说，如果我们想计算出 Thor 的中介中心度，首先要计算 Thor 之外的所有用户对之间的最短路径，然后再统计有多少条最短路径通过了 Thor 这个节点。比如说，Chi（id 为 3) 和 Clive（id 为 5) 之间唯一的最短路径经过了 Thor 这个节点，而 Hero（id 为 0) 和 Chi（id 为 3) 之间的两条最短路径则都没有经过 Thor。

因此，作为第一步，我们需要找出所有用户对之间的最短路径。不过，尽管存在许多可以高效计算最短路径的尖端算法，但是按照我们的惯例，这里将采用效率虽低一些但更加容易理解的算法。

这个算法（一个广度优先搜索的实现）在本书中算是比较复杂的一种，所以我们仔细探讨。

1. 我们的目标是建立一个以 from_user 为输入的函数，它能够找出到达其他每个用户的所有最短路径。
2. 我们将通过用户 ID 组成的列表来表示路径。由于每条路径的第一个节点总是 from_user，因此我们可以在这个列表中将该 ID 忽略。也就是说，这个代表路径的列表的长度等于该路径本身的长度。
3. 我们将维护一个名为 shortest_paths_to 的字典，其键为用户 ID，其值为以该用户 ID 结尾的路径构成的列表。如果最短路径是唯一的，那么这个列表就只包含一个路径。如果有多条最短路径的话，那么该列表将包含所有这些路径。
4. 我们还将维护一个名为 frontier 的队列来存放那些待考察的用户，并且它们的存放顺序就是相应的考察顺序。我们将以用户对的形式——即 (prev_user, user)——来进行存储，这样就能了解我们是如何到达每一个用户的。这个队列是通过 from_user 的所有相邻节点进行初始化的。（当然，我们之前从没有讨论过队列，其实它是专门为“在后端进行插入”操作和“在前端进行删除”操作经过优化的数据结构。在 Python 语言中，队列是由 collections.deque 模块来实现的，实际上它是一种双向队列。）
5. 当我们在图中进行探索的时候，每当发现新的邻居节点，只要还不知道通向它们的最短路径，就将它们添加到队尾以供将来进一步探索，并且以当前用户作为其 prev_user。
6. 当我们把一个用户从队列中删除时，如果之前从未遇到过该用户，那么我们肯定是找到了一个或多个通向它的最短路径：沿着到达 prev_user 的每个最短路径再走一步即是。
7. 当我们从队列中删除一个之前遇到过的用户时，我们不是找到了另一个最短路径（这种情况下我们应该将其添加到队尾），就是找到了一个更长的路径（这种情况下不用将其插入队尾）。
8. 当队列中已经没有用户时，说明我们已经搜遍了整个图（或者至少也是起始用户所能够达到的部分），这时我们就可以停止了。

我们可以将这些步骤放入一个（大型）函数中，代码如下所示：

```
from collections import deque

def shortest_paths_from(from_user):
```

```
    # 一个由"user_id"到该用户所有最短路径的字典
    shortest_paths_to = { from_user["id"] : [[]] }

    # 我们需要检查的(previous user, next user)队列
    # 从所有(from_user, friend_of_from_user)对开始着手

    frontier = deque((from_user, friend)
                     for friend in from_user["friends"])

    # 直到队列为空为止
    while frontier:

        prev_user, user = frontier.popleft()   # 删除该用户
        user_id = user["id"]                   # 即队列中的第一个用户

        # 若要向队列添加内容
        # 我们必须知道通向prev_user的某些最短路径
        paths_to_prev_user = shortest_paths_to[prev_user["id"]]
        new_paths_to_user = [path + [user_id] for path in paths_to_prev_user]

        # 我们可能已经知道了一条最短路径
        old_paths_to_user = shortest_paths_to.get(user_id, [])

        # 到目前为止,我们看到的到达这里的最短路径有多长?
        if old_paths_to_user:
            min_path_length = len(old_paths_to_user[0])
        else:
            min_path_length = float('inf')

        # 只留下那些刚找到的不太长的路径
        new_paths_to_user = [path
                             for path in new_paths_to_user
                             if len(path) <= min_path_length
                             and path not in old_paths_to_user]

        shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user

        # 将这些从未谋面的"邻居"添加到frontier中
        frontier.extend((user, friend)
                        for friend in user["friends"]
                        if friend["id"] not in shortest_paths_to)

    return shortest_paths_to
```

现在我们可以将这些 dict 存放到各个节点中了：

```
for user in users:
    user["shortest_paths"] = shortest_paths_from(user)
```

好了，现在终于可以计算中介中心度了。对于任意一对节点 i 和 j，我们都知道从 i 到 j 有 n 条最短路径。然后，对应于每一条这样的最短路径，我们只要给该路径中的每个节点的中心度加 $1/n$ 即可：

```
for user in users:
    user["betweenness_centrality"] = 0.0

for source in users:
    source_id = source["id"]
    for target_id, paths in source["shortest_paths"].iteritems():

        if source_id < target_id:      # 不要加倍计数
            num_paths = len(paths)     # 有多少最短路径
            contrib = 1 / num_paths    # 中心度加1/n
            for path in paths:
                for id in path:
                    if id not in [source_id, target_id]:
                        users[id]["betweenness_centrality"] += contrib
```

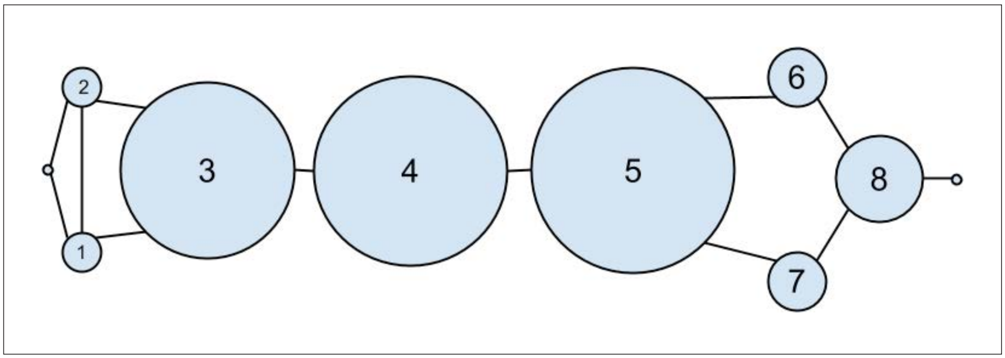
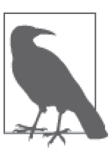


图 21-2：根据中介中心度的大小绘制的 DataSciencester 网络

如图 21-2 所示，用户 0 和 9 的中心度为 0（因为它们不在其他用户之间的任何一条最短路径上），而用户 3、4 和 5 则都具有较高的中心度（因为它们都位于多条最短路径上）。



一般来说，中心度的数值本身并不具有多大意义。我们关心的是每个节点的中心度数值与其他节点的相对大小。

此外，还有一个需要关注的衡量指标，即所谓的*接近中心度*（closeness centrality）。首先，为每个用户计算其*疏远度*（farness），即该用户到所有其他用户的最短路径的长度总和。

由于我们已经计算出每一对节点之间的最短路径，因此，只要对其求和即可。（如果有多个最短路径，并且都具有相同的长度，那么我们只考察第一个即可。）

```
def farness(user):
    """the sum of the lengths of the shortest paths to each other user"""
    return sum(len(paths[0])
               for paths in user["shortest_paths"].values())
```

这样一来，接近中心度的计算量就很小了（见图 21-3）：

```
for user in users:
    user["closeness_centrality"] = 1 / farness(user)
```

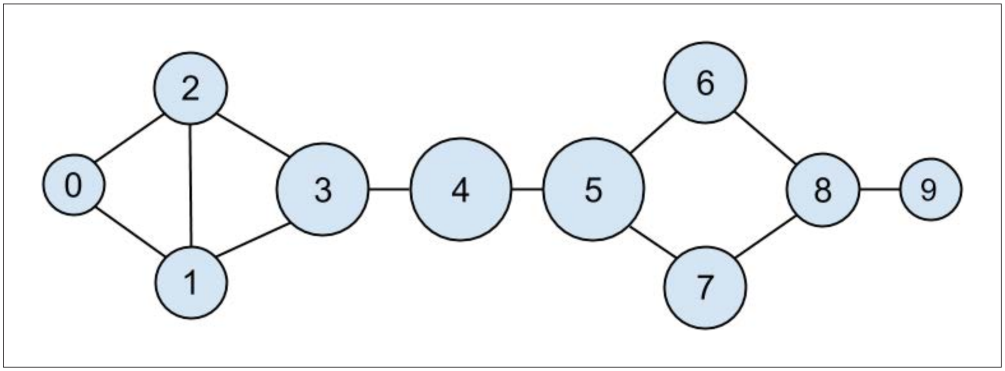


图 21-3：根据接近中心度绘制的 DataSciencester 网络

我们看到，尽管非常靠近中心的节点离外围节点很远，但是图中各节点在大小上面差别不是很大。

正如我们所看到的那样，计算最短路径是一件苦差事。因此，中介中心度和接近中心度很少用于大型网络。还有一种不太直观的特征向量中心度（eigenvector centrality），由于计算起来更容易，所以更加常用。

21.2 特征向量中心度

要想介绍特征向量中心度，我们就不得不讨论特征向量，而要想讨论特征向量，我们就必须介绍矩阵乘法。

21.2.1 矩阵乘法

假设 A 是一个 $n_1 \times k_1$ 矩阵，B 是一个 $n_2 \times k_2$ 矩阵，并且 $k_1 = n_2$，那么这两个矩阵的乘积 AB 则是一个 $n_1 \times k_2$ 矩阵，其中第 (i,j) 项为：

$$\mathrm{A}_{i1}\mathrm{B}_{1j} + \mathrm{A}_{i2}\mathrm{B}_{2j} + \cdots + \mathrm{A}_{ik}\mathrm{B}_{kj}$$

下面，我们仅计算矩阵 A 的第 i 行（可以视为一个向量）与矩阵 B 的第 j 列（也可以视为一个向量）的点积，具体代码如下所示：

```
def matrix_product_entry(A, B, i, j):
    return dot(get_row(A, i), get_column(B, j))
```

之后，我们就可以通过下列代码实现矩阵的乘法运算了：

```
def matrix_multiply(A, B):
    n1, k1 = shape(A)
    n2, k2 = shape(B)
    if k1 != n2:
        raise ArithmeticError("incompatible shapes!")

    return make_matrix(n1, k2, partial(matrix_product_entry, A, B))
```

请注意，如果 A 是一个 $n \times k$ 矩阵，B 是一个 $k \times 1$ 矩阵，那么 AB 则为 $n \times 1$ 矩阵。如果我们将一个向量看作是一维矩阵，就可以把 A 看作是将 k 维向量映射为 n 维向量的一个函数，实际上这个函数就是矩阵乘法。

之前，我们将向量简单地表示为列表，实际上两者之间是不能完全划等号的：

```
v = [1, 2, 3]
v_as_matrix = [[1],
               [2],
               [3]]
```

因此，我们需要定义相应的辅助函数，以便实现两种表示形式之间的转换：

```
def vector_as_matrix(v):
    """returns the vector v (represented as a list) as a n x 1 matrix"""
    return [[v_i] for v_i in v]

def vector_from_matrix(v_as_matrix):
    """returns the n x 1 matrix as a list of values"""
    return [row[0] for row in v_as_matrix]
```

如此一来，我们就可以利用 matrix_multiply 来定义矩阵运算了：

```
def matrix_operate(A, v):
    v_as_matrix = vector_as_matrix(v)
    product = matrix_multiply(A, v_as_matrix)
    return vector_from_matrix(product)
```

当 A 是一个方阵时，此操作会将 n 维向量映射为另一个 n 维向量。对于某矩阵 A 和向量 v，对向量 v 进行 A 变换有时候会等效于用一个标量来乘向量 v，即所得到的向量与 v 同向。当发生这种情况（且 v 不是零向量）时，我们称 v 为 A 的特征向量。同时，我们称这个乘数为特征值（eigenvalue）。

确定矩阵 A 的特征向量的一种可行方法是取一个随机向量 v，然后利用 `matrix_operate` 对其进行调整，从而得到一个长度为 1 的向量，重复该过程直到收敛为止：

```
def find_eigenvector(A, tolerance=0.00001):
    guess = [random.random() for __ in A]
```

```
    while True:
        result = matrix_operate(A, guess)
        length = magnitude(result)
        next_guess = scalar_multiply(1/length, result)

        if distance(guess, next_guess) < tolerance:
            return next_guess, length   # eigenvector, eigenvalue
            guess = next_guess
```

通过这种构造方法返回的向量 `guess` 将具备这样的特点：当你对它应用 `matrix_operate` 函数并将其长度缩为 1 的时候，得到的向量与其自身极为接近。这就意味着它是一个特征向量。

请注意，并不是所有的实数矩阵都具有特征向量和特征值。例如，请看下列矩阵：

```
rotate = [[ 0, 1],
          [-1, 0]]
```

上述代码的作用是按照顺时针方向将向量旋转 90 度，这意味着，对于这个矩阵来说，只有一个向量能够映射到自身的数乘上面，这个向量就是零向量。如果你执行 `find_eigenvector(rotate)`，它会永远运行下去。即使是具备特征向量的矩阵，有时候也会陷入这种死循环。请看下面的矩阵：

```
flip = [[0, 1],
        [1, 0]]
```

对于任意向量 `[x, y]`，这个矩阵都会将其映射为 `[y, x]`。这就意味着，`[1, 1]` 是一个特征值为 1 的特征向量。但是，如果你从一个 x 和 y 并不相等的随机向量着手的话，那么，`find_eigenvector` 将会来回交换这两个坐标值，并且永远也不会停下来。（Not-from-scratch 是一个类似于 NumPy 的 Python 库，由于它采用了不同的处理方法，因此能够有效处理这种情形。）尽管如此，只要 `find_eigenvector` 能返回一个结果，那么这个结果肯定是一个特征向量。

21.2.2 中心度

该如何利用特征向量来帮助我们理解 DataSciencester 网络呢？

首先，我们需要用 `adjacency_matrix` 来表示网络中的连接，其中第 (i,j) 个元素的值要么为 1（如果用户 i 和用户 j 是朋友的话），要么为 0（如果他们不是朋友的话）：

```
def entry_fn(i, j):
    return 1 if (i, j) in friendships or (j, i) in friendships else 0

n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

对于每个用户来说，他的特征向量中心度就是在 `find_eigenvector` 返回的本征向量中的该用户对应的那个元素（见图 21-4）：

具体技术细节这里就不多说了，大家只要知道任何非零的邻接矩阵必然有一个所有的值皆非负的特征向量就行了。幸运的是，我们只要借助于 find_eigenvector 函数就能够找到这种 adjacency_matrix。

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

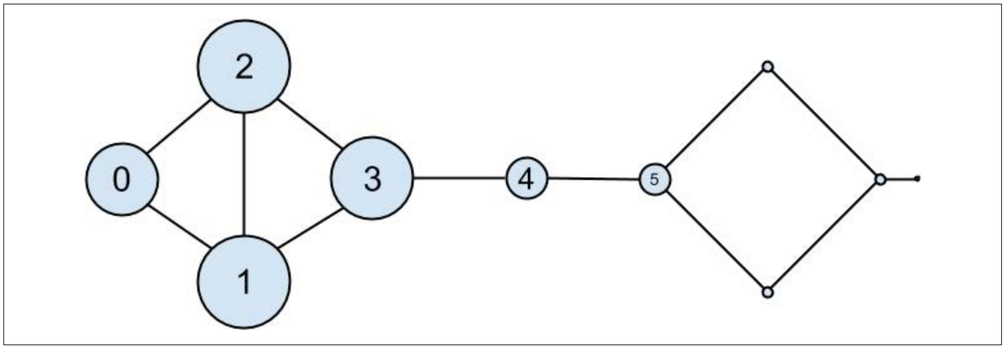


图 21-4：根据特征向量中心度绘制的 DataSciencester 网络

特征向量中心度较高的用户，不仅会拥有较多的连接，而且还倾向于连接到具有较高中心度的那些人。

就上图而言，用户 1 和用户 2 具有最高的中心度，这是由于他们两个都有三条连接是通向具有高中心度的对方的。如果我们将其移除，他们的中心度就会直线下降。

在这种小型的网络上中，特征向量中心度的行为会有些怪异。当你尝试增减连接的时候，你会发现，只要对网络进行稍微的修改，中心度的数值就会发生戏剧性的变化。对于比较大型的网络来说，这种情况就不太明显。

我们仍然没有介绍为什么特征向量能够较好地度量中心度。这是因为特征向量意味着，如果你计算：

```
matrix_operate(adjacency_matrix, eigenvector_centralities)
```

其结果就等于用一个标量去乘以 eigenvector_centralities。

如果你了解矩阵乘法的运算机制，就会知道 matrix_operate 求出的向量的第 i 个元素为：

```
dot(get_row(adjacency_matrix, i), eigenvector_centralities)
```

这实际上就是对连接到用户 i 的各个用户的特征向量中心度进行求和。

换句话说，特征向量中心度就是一些数值，即每个用户对应一个数值，而每个用户的值就是他的相邻值之和的固定倍数。在这种情况下，中心度就意味着要跟处于中心地位的人交

朋友。你结交的人的中心度越高，你的中心度也就越高。当然，这明显是一个循环定义，而特征向量就是打破这个循环的突破口。

对此还有另外一种理解方法，那就是考察 `find_eigenvector` 函数的处理方式。它首先给每个节点随机指定一个中心度，然后重复以下两个步骤，直到这个过程收敛为止。

1. 赋予每个节点一个新的中心度分数，该分数等于该节点相邻节点的（原）中心度分数之和。
2. 调整中心度向量，直到其大小变成 1 为止。尽管这种做法包含的数学原理有点让人摸不着头脑，但是就计算本身而言，还是相当简单的（这一点与中介中心度不同），同时，它也非常适用于巨型网络图。

21.3　有向图与PageRank

由于 DataSciencester 没有获得人们的热烈追捧，因此，负责营收的副总决定将网站从交友模式转换为赞助模式。事实证明，除了高科技业的猎头非常关心哪些数据科学家备受其他数据科学家推崇之外，没人对科学家之间的好友关系特别在意。

在这个新的模型中，我们所关注的赞助 `(source, target)` 并不表示互反关系，而是表示用户 `source` 认为用户 `target` 是一位令人惊畏的数据科学家（见图 21-5）。因此，我们需要考虑这种不对称性：

```
endorsements = [(0, 1), (1, 0), (0, 2), (2, 0), (1, 2),
                (2, 1), (1, 3), (2, 3), (3, 4), (5, 4),
                (5, 6), (7, 5), (6, 8), (8, 7), (8, 9)]

for user in users:
    user["endorses"] = []       # 增加一个列表来追踪外方的赞助
    user["endorsed_by"] = []    # 增加另外一个列表来追踪赞助

for source_id, target_id in endorsements:
    users[source_id]["endorses"].append(users[target_id])
    users[target_id]["endorsed_by"].append(users[source_id])
```

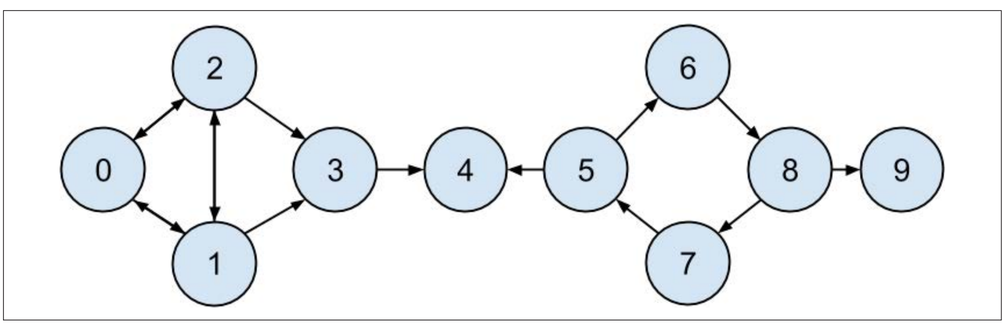


图 21-5：基于赞助关系的 DataSciencester 网络

这样的话，我们就能够轻而易举地找出 most_endorsed（最受推崇的）数据科学家，从而将这些信息出售给猎头们：

```
endorsements_by_id = [(user["id"], len(user["endorsed_by"]))
                      for user in users]

sorted(endorsements_by_id,
       key=lambda (user_id, num_endorsements): num_endorsements,
       reverse=True)
```

然而，“赞同票数”这种指标是很容易被人搞鬼的。实际上，你只要创建大量傀儡账户，然后让这些账户给你投票就行了。或者，你还可以跟朋友们商量好，都彼此捧场也行。（例如用户 0、1 和 2 好像就是这么干的。）

因此，指标最好还要考虑到给你投赞同票的那些人。也就是说，来自得票数较多的人的投票的分量应该重于得票数较少的那些人的投票。这实际上就是 PageRank 算法的思想精华，Google 就是利用它来给网站排名的，主要考量的就是链接到该网站的其他站点、到达该网站的链接等。

（这是否让你想起了特征向量中心度背后的思想依据呢？）

下面是这种思想的简化版本。

1. 网络中 PageRank 的总分数为 1（或 100%）。
2. 最初，这个 PageRank 被均匀分布到网络的各个节点中。
3. 在每一步中，每个节点的 PageRank 很大一部分将均匀分布到其外部链接中。
4. 在每个步骤中，每个节点的 PageRank 的其余部分被均匀地分布到所有节点上。

```
def page_rank(users, damping = 0.85, num_iters = 100):

    # 一开始均匀分布PageRank
    num_users = len(users)
    pr = { user["id"] : 1 / num_users for user in users }

    # 这是PageRank的一小部分
    # 每个节点进行各自的迭代
    base_pr = (1 - damping) / num_users

    for __ in range(num_iters):
        next_pr = { user["id"] : base_pr for user in users }
        for user in users:
            # 将PageRank分布到外部链接中
            links_pr = pr[user["id"]] * damping
            for endorsee in user["endorses"]:
                next_pr[endorsee["id"]] += links_pr / len(user["endorses"])
        pr = next_pr

    return pr
```

PageRank（见图 21-6）表明，用户 4（也就是 Thor）是排名最高的数据科学家。

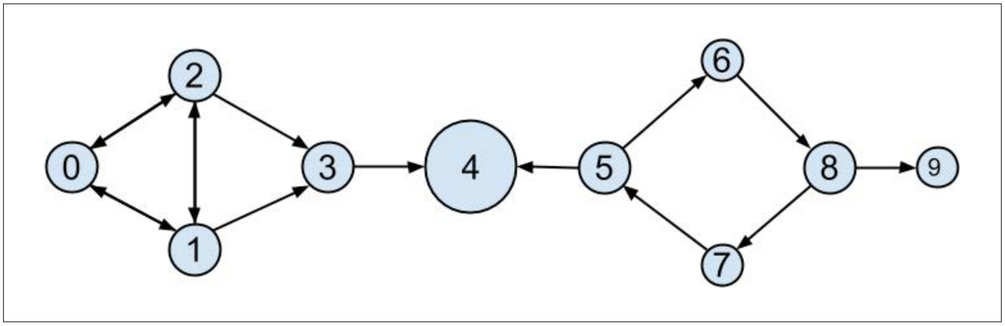


图 21-6：利用 PageRank 绘制的 DataSciencester 网络

与用户 0、1 和 2 相比，虽然给他投票的人（2 个）并不多，但是他的得票数还要考虑投票方自身的排名。此外，两个投票方都给只给他投了票，这就意味着他不必与别人分享他们的排名。

21.4　延伸学习

- 除了本文介绍的这些中心度指标外，还有许多其他不同的指标（https://en.wikipedia.org/wiki/Centrality），不过这里所介绍的都是些最常见的指标。
- NetworkX（http://networkx.github.io/）是一个用于网络分析的 Python 库。它为我们提供了许多函数，来帮助计算中心度以及实现图的可视化。
- Gephi（http://gephi.github.io/）是一个让人爱恨交织的基于图形用户界面的网络可视化工具。

第22章

推荐系统

> 哦，老天，老天，你为什么如此随便，老是把提供错误建议的人送到这个世间？！
>
> ——亨利 · 菲尔丁

现实中，利用数据提供某种建议也是很常见的。例如，Netflix 能够向用户推荐他们可能想看的电影，亚马逊会向你推荐你可能会买的商品，Twitter 会为你推荐你可能想关注的用户。本章将介绍几种利用数据来提供建议的方法。

需要指出的是，这里要考察的 `users_interests` 是之前就曾用过的一个数据集：

```
users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

同时，我们要考虑如何根据用户当前特定的兴趣来向其推荐新的感兴趣的东西。

22.1 手工甄筛

在互联网出现之前，如果你需要得到某些读书建议的话，恐怕得到图书馆去，那里的图书管理员通常能够根据你的兴趣或者你喜欢的书籍来推荐书籍。

鉴于 DataSciencester 的用户及其兴趣的数量有限，你只需花费一个下午的时间就能轻松通过人工方式向每个用户推荐他们感兴趣的东西。但这种方法并不是特别好，因为它受制于你的个人经验和想象力。（当然，这并不意味着我认为你的个人经验和想象力是有限的。）因此，我们要设法让数据来做这件事情。

22.2 推荐流行事物

一个比较简单的方法就是直接推荐比较流行的东西：

```
popular_interests = Counter(interest
                            for user_interests in users_interests
                            for interest in user_interests).most_common()
```

得到的结果可能像下面这样：

```
[('Python', 4),
 ('R', 4),
 ('Java', 3),
 ('regression', 3),
 ('statistics', 3),
 ('probability', 3),
 # …
]
```

完成上述计算后，我们就可以向用户推荐那些当前最流行的、他尚未感兴趣的东西：

```
def most_popular_new_interests(user_interests, max_results=5):
    suggestions = [(interest, frequency)
                   for interest, frequency in popular_interests
                   if interest not in user_interests]
    return suggestions[:max_results]
```

因此，如果你是用户 1，并且当前的兴趣为：

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

那么，我们会向你推荐下列内容：

```
most_popular_new_interests(users_interests[1], 5)

# [('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics', 3)]
```

如果你是用户 3，而且上面这些东西中有很多早就是你之前的兴趣所在了，那么你会得到

下面的建议：

```
[('Java', 3),
 ('HBase', 3),
 ('Big Data', 3),
 ('neural networks', 2),
 ('Hadoop', 2)]
```

当然，“很多人对 Python 感兴趣，所以你也可能感兴趣”并非最具有说服力的推销口号。如果某人是我们网站的新人，我们对他一无所知，那么这可能就是最好的推荐方法了。下面让我们看看如何根据用户的兴趣来提供更好的建议。

22.3 基于用户的协同过滤方法

一种利用用户兴趣的方法是根据这些兴趣找到有*类似爱好*的人，然后再根据这些人的爱好来向你推荐你可能感兴趣的东西。

为此，我们需要找到一种指标来衡量两个用户之间的相似程度。就这里来说，我们所使用的指标叫作*余弦相似度*（cosine similarity）。给定两个向量 v 和 w，余弦相似度的定义如下所示：

```
def cosine_similarity(v, w):
    return dot(v, w) / math.sqrt(dot(v, v) * dot(w, w))
```

它用来测量 v 和 w 之间的“角度”。如果 v 和 w 指向同一个方向，那么分子和分母都相等，所以其余弦相似度等于 1。如果 v 和 w 指向相反的方向，那么其余弦相似度就等于 -1。如果 v 等于 0，那么无论 w 是否为 0（反之亦然），dot(v, w) 总是等于 0，所以余弦相似度总为 0。

我们会把这个计算方法应用到由 0 和 1 构成的向量上面，其中每个向量都表示一个用户的某种爱好。如果用户对第 *i* 项事物感兴趣的话，则 v[i] 取 1，否则为 0。因此，“爱好相似的用户”就意味着“兴趣向量的方向几乎相同的用户”。兴趣完全相同的用户，其相似度为 1。而没有共同兴趣的用户，其相似度为 0。否则的话，其相似度会介于两者之间：该数值越接近 1，表示“越相似”，该数值越接近 0，表示“越不相似”。

通常来说，从收集已知的兴趣并为其（隐式）指定索引着手是一个不错的主意。为此，我们可以用集合的观点将那些不重复的兴趣收集到一起，然后把它们放到一个列表中，并对其进行排序。这样的话，在这个列表中的第一个兴趣就是兴趣 0，其他以此类推：

```
unique_interests = sorted(list({ interest
                                 for user_interests in users_interests
                                 for interest in user_interests }))
```

我们将得到一个列表，开头部分如下所示：

```
['Big Data',
 'C++',
 'Cassandra',
 'HBase',
 'Hadoop',
 'Haskell',
 # …
]
```

接下来，我们要给每个用户生成一个由 0 和 1 组成的“兴趣”向量。为此，我们只需遍历 `unique_interests` 列表，如果用户有某种兴趣，则相应元素置 1，否则置 0：

```
def make_user_interest_vector(user_interests):
    """given a list of interests, produce a vector whose ith element is 1
    if unique_interests[i] is in the list, 0 otherwise"""
    return [1 if interest in user_interests else 0
            for interest in unique_interests]
```

之后，我们还可以创建用户兴趣矩阵，为此，我们只需将这个函数映射到由用户的兴趣构成的列表的列表上面即可：

```
user_interest_matrix = map(make_user_interest_vector, users_interests)
```

现在，如果用户 `i` 对兴趣 `j` 感兴趣，则 `user_interest_matrix[i][j]` 等于 1，否则为 0。

由于我们的数据集非常小，因此所有用户两两之间的相似性的计算量不是很大：

```
user_similarities = [[cosine_similarity(interest_vector_i, interest_vector_j)
                      for interest_vector_j in user_interest_matrix]
                     for interest_vector_i in user_interest_matrix]
```

在这之后，`user_similarities[i][j]` 的数值就能够告诉我们用户 i 和用户 j 之间的相似度。

举例来说，`user_similarities[0][9]` 等于 0.57，因为这两个用户都对 Hadoop、Java 和 Big Data 感兴趣。同时，`user_similarities[0][8]` 的值只有 0.19，因为用户 0 和用户 8 只有一个共同的兴趣，即 Big Data。

就 `user_similarities[i]` 而言，它存放的是用户 `i` 相对于所有用户的相似度。我们可以用它来写一个函数，来找出与给定用户最相似的用户。同时，我们还需要确保这里不包括用户自身以及相似度为 0 的那些用户。下面我们按照相似度从大到小的顺序对结果进行排序：

```
def most_similar_users_to(user_id):
    pairs = [(other_user_id, similarity)                      # 查找
             for other_user_id, similarity in                 # 其他用户
                enumerate(user_similarities[user_id])         # 非零
             if user_id != other_user_id and similarity > 0]  # 相似度

    return sorted(pairs,                                      # 将其排序
                  key=lambda (_, similarity): similarity,     # 相似度
                  reverse=True)                               # 由大到小
```

举例来说，如果我们调用函数 most_similar_users_to(0)，将得到下列输出：

```
[(9, 0.5669467095138409),
 (1, 0.3380617018914066),
 (8, 0.1889822365046136),
 (13, 0.1690308509457033),
 (5, 0.1543033499620919)]
```

那么，我们该如何利用这个结果向用户推荐新的兴趣呢？对于每种兴趣，我们可以将其他对其感兴趣的用户的用户相似度加起来：

```
def user_based_suggestions(user_id, include_current_interests=False):
    # 将相似度加起来
    suggestions = defaultdict(float)
    for other_user_id, similarity in most_similar_users_to(user_id):
        for interest in users_interests[other_user_id]:
            suggestions[interest] += similarity

    # 将它们转化成已排序的列表
    suggestions = sorted(suggestions.items(),
                         key=lambda (_, weight): weight,
                         reverse=True)

    # 并且(有可能)排除已存在的兴趣
    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

如果我们调用 user_based_suggestions(0)，那么在推荐的兴趣中比较靠前的几个为：

```
[('MapReduce', 0.5669467095138409),
 ('MongoDB', 0.50709255283711),
 ('Postgres', 0.50709255283711),
 ('NoSQL', 0.3380617018914066),
 ('neural networks', 0.1889822365046136),
 ('deep learning', 0.1889822365046136),
 ('artificial intelligence', 0.1889822365046136),
 #...
]
```

这些东西对于自称对“Big Data”和数据库相关的主题感兴趣的人来说，看起来的确是相当不错的建议。（这些权重自身并无意义，它们只是用于进行排序。）

但是，当兴趣的数量很大的时候，这种方法就玩不转了。还记得第 12 章中介绍的维度灾难吗？在高维向量空间中，绝大多数向量之间都是离得非常远的（因此它们之间的方向也悬殊很大）。也就是说，当兴趣的数量变大时，即使是与给定用户“最相似的用户”，实际上也很可能根本没有相似之处。

想象一个类似亚马逊这样的网站，在过去几十年中，我已经从它那里购买了数以千计的商品。你可能试图通过购买模式来找出跟我类似的用户，但在这个世界上，除了我自己之外，恐怕没有谁的购买历史看起来更像我了。无论与我“最相似的”顾客是谁，他很可能根本就不像我，因此，如果将他购买的物品推荐给我的话，基本上就注定了这是一个糟糕的建议。

22.4　基于物品的协同过滤算法

还有一种方法，即直接计算两种兴趣之间的相似度，然后将与用户当前兴趣相似的兴趣放到一起，并从中为用户推荐感兴趣的东西。

首先，我们要对用户兴趣矩阵进行转置（transpose），以使行对应于兴趣，列对应于用户：

```
interest_user_matrix = [[user_interest_vector[j]
                         for user_interest_vector in user_interest_matrix]
                        for j, _ in enumerate(unique_interests)]
```

这么做之后，结果如何呢？这时，矩阵 `interest_user_matrix` 的行 `j` 就是矩阵 `user_interest_matrix` 的列 `j`。也就是说，1 表示用户有某种兴趣，0 表示用户没有某种兴趣。

举例来说，假设 `unique_interests[0]` 为 Big Data，那么 `interest_user_matrix[0]` 则为：

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

这是因为用户 0、8 和 9 都对 Big Data 感兴趣。

现在，我们可以再次利用余弦相似度。如果喜欢两个主题的用户完全重合，那么它们的相似度为 1。如果喜欢两个主题的用户没有一个是重合的，那么它们的相似度将是 0：

```
interest_similarities = [[cosine_similarity(user_vector_i, user_vector_j)
                          for user_vector_j in interest_user_matrix]
                         for user_vector_i in interest_user_matrix]
```

例如，我们可以通过下列代码找出与 Big Data（兴趣 0）最相似的项：

```
def most_similar_interests_to(interest_id):
    similarities = interest_similarities[interest_id]
    pairs = [(unique_interests[other_interest_id], similarity)
             for other_interest_id, similarity in enumerate(similarities)
             if interest_id != other_interest_id and similarity > 0]
    return sorted(pairs,
                  key=lambda (_, similarity): similarity,
                  reverse=True)
```

下面是推荐的相似兴趣：

```
[('Hadoop', 0.8164965809277261),
 ('Java', 0.6666666666666666),
 ('MapReduce', 0.5773502691896258),
 ('Spark', 0.5773502691896258),
 ('Storm', 0.5773502691896258),
 ('Cassandra', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('neural networks', 0.4082482904638631),
 ('HBase', 0.3333333333333333)]
```

现在，我们可以通过总结与其兴趣相似的东西来为其提供建议：

```
def item_based_suggestions(user_id, include_current_interests=False):
    # 将相似的兴趣相加
    suggestions = defaultdict(float)
    user_interest_vector = user_interest_matrix[user_id]
    for interest_id, is_interested in enumerate(user_interest_vector):
        if is_interested == 1:
            similar_interests = most_similar_interests_to(interest_id)
            for interest, similarity in similar_interests:
                suggestions[interest] += similarity

    # 根据权重将其排序
    suggestions = sorted(suggestions.items(),
                         key=lambda (_, similarity): similarity,
                         reverse=True)

    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

下面是为用户 0 提供的（看上去比较恰当的）建议：

```
[('MapReduce', 1.861807319565799),
 ('Postgres', 1.3164965809277263),
 ('MongoDB', 1.3164965809277263),
 ('NoSQL', 1.2844570503761732),
 ('programming languages', 0.5773502691896258),
 ('MySQL', 0.5773502691896258),
 ('Haskell', 0.5773502691896258),
 ('databases', 0.5773502691896258),
 ('neural networks', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('C++', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('Python', 0.2886751345948129),
 ('R', 0.2886751345948129)]
```

22.5 延伸学习

- Crab（http://muricoca.github.io/crab/）是一个打造推荐系统的 Python 框架。
- Graphlab 也提供了一个推荐工具包（https://dato.com/products/create/docs/graphlab.toolkits.recommender.html）。
- Netflix Prize（http://www.netflixprize.com/）是一项比较著名的竞赛活动，旨在为 Netflix 用户打造更好的电影推荐系统。

第23章

数据库与SQL

回忆，是最贴心的朋友，也是最可怕的敌人。

——吉尔伯特·帕克

我们使用的数据通常存储于数据库中。数据库是用来有效存储与查询数据的系统。绝大部分数据库属于关系型（relational）数据库，例如 Oracle、MySQL 与 SQL Server，它们把数据存储在表中，并专门通过结构化查询语言（SQL）来查询。SQL 是一种用来处理数据的声明性语言。

SQL 是数据科学家工具箱中相当重要的一部分。本章中我们来创建 NotQuiteABase——一个类似于数据库的 Python 实现。我们会学习 SQL 的基础知识，并且在我们的类数据库中展示它是如何工作的，这是让你从零基础开始理解它们工作原理的最佳方式。希望你通过解决 NotQuiteABase 中的问题，可以很好地理解在 SQL 中如何解决相同的问题。

23.1 CREATE TABLE与INSERT

关系型数据库是表（以及表之间的关系）的集合。表是行的简单集合，这与我们前面讨论过的矩阵不同。但是表有一个固定的结构，包含列名与列的类型。

例如，假设有一个数据集 `users`，对于每个用户它都包含三个属性 `user\_id`、`name` 和 `num\_friends`：

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
```

```
        [2, "Sue", 3],
        [3, "Chi", 3]]
```

在 SQL 中，我们可以这样创建表：

```
CREATE TABLE users (
    user_id INT NOT NULL,
    name VARCHAR(200),
    num_friends INT);
```

注意，我们设定 `user\_id` 和 `num\_friends` 都必须是整数（且 `user_id` 不可以为 `NULL`，因为 `NULL` 表示缺失值，类似于我们的 `None`），并且名字必须是长度不大于 200 的字符串。NotQuiteABase 不考虑类型，但这里我们当作它会考虑。

> SQL 几乎不区分大小写和缩进形式。本书中的大小写与缩进风格是我的个人习惯。如果你开始学习 SQL，很可能会遇上不同风格的例子。

你可以使用 `INSERT` 语句来插入行：

```
INSERT INTO users (user_id, name, num_friends) VALUES (0, 'Hero', 0);
```

注意，SQL 语句需要用分号结尾，并且字符串需要用单引号括住。

在 NotQuiteABase 中，只要简单地设定列名就能创建一个表。要想插入一个行，你可以使用表的 `insert()` 方法，这个方法使用一个包含行值的列表，行值需按照表的列名顺序排列。

从本质上说，我们存储的每行都类似于一个从列名到值映射的字典。一个真正的数据库不会使用这么浪费空间的方式，但这样做可以使 NotQuiteABase 更易于处理：

```
class Table:
    def __init__(self, columns):
        self.columns = columns
        self.rows = []

    def __repr__(self):
        """pretty representation of the table: columns then rows"""
        return str(self.columns) + "\n" + "\n".join(map(str, self.rows))

    def insert(self, row_values):
        if len(row_values) != len(self.columns):
            raise TypeError("wrong number of elements")
        row_dict = dict(zip(self.columns, row_values))
        self.rows.append(row_dict)
```

例如，我们可以创建：

```
users = Table(["user_id", "name", "num_friends"])
users.insert([0, "Hero", 0])
users.insert([1, "Dunn", 2])
users.insert([2, "Sue", 3])
users.insert([3, "Chi", 3])
users.insert([4, "Thor", 3])
users.insert([5, "Clive", 2])
users.insert([6, "Hicks", 3])
users.insert([7, "Devin", 2])
users.insert([8, "Kate", 2])
users.insert([9, "Klein", 3])
users.insert([10, "Jen", 1])
```

如果你打印 users，可以看到：

```
['user_id', 'name', 'num_friends']
{'user_id': 0, 'name': 'Hero', 'num_friends': 0}
{'user_id': 1, 'name': 'Dunn', 'num_friends': 2}
{'user_id': 2, 'name': 'Sue', 'num_friends': 3}
...
```

23.2 UPDATE

有时候你需要更新已经存在于数据库中的数据。例如，如果 Dunn 结识了一位新朋友，你就得这样做：

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

其核心特征是：

- 哪张表需要更新
- 哪些行需要更新
- 哪些字段需要更新
- 新值应该是什么

我们将为 NotQuiteABase 增加类似的 update 方法。第一个语句是 dict，其键是需要更新的列，其值是这些字段的新值。第二个语句是 predicate，它对需要更新的行返回 True，否则返回 False：

```
def update(self, updates, predicate):
    for row in self.rows:
        if predicate(row):
            for column, new_value in updates.iteritems():
                row[column] = new_value
```

然后我们只需这样做：

```
users.update({'num_friends' : 3},              # 设定 num_friends = 3
             lambda row: row['user_id'] == 1)  # 在user_id == 1的行中
```

23.3 DELETE

SQL 有两种方法可以在表中删除行。一个比较有风险的方法是直接删掉表的每行：

```
DELETE FROM users;
```

稍微安全一点的方法是增加 WHERE 子句，再删掉匹配某种条件的行：

```
DELETE FROM users WHERE user_id = 1;
```

在表中增加这个功能很容易：

```
    def delete(self, predicate=lambda row: True):
        """delete all rows matching predicate
        or all rows if no predicate supplied"""
        self.rows = [row for row in self.rows if not(predicate(row))]
```

如果你提供了一个 predicate 函数（即 WHERE 子句），它会仅删掉那些满足条件的行。如果你不提供任何 predicate 函数，默认的判定就返回真值 True，然后删掉每一行。

例如：

```
users.delete(lambda row: row["user_id"] == 1)  # 删掉user_id == 1的行
users.delete()                                 # 删掉每一行
```

23.4 SELECT

通常，我们不会直接查看 SQL 表，而是通过一个 SELECT 语句查询：

```
SELECT * FROM users;                            -- 得到所有内容
SELECT * FROM users LIMIT 2;                    -- 得到前两行
SELECT user_id FROM users;                      -- 只得到特定列
SELECT user_id FROM users WHERE name = 'Dunn';  -- 只得到特定行
```

你也可以使用 SELECT 语句计算字段：

```
SELECT LENGTH(name) AS name_length FROM users;
```

我们会给 Table 类一个 select() 方法来返回一个新 Table。这个方法采用两种可选语句。

- keep_columns 声明了你希望在结果中保留的列名。如果没提供这一项，结果会包含所有的列。

- additional_columns 是一个字典，它的键是新列名，值是指定如何计算新列值的函数。

如果两种都不用，你只会得到一个表的机械复制的版本：

```
def select(self, keep_columns=None, additional_columns=None):

    if keep_columns is None:         # 如果没有指定列
        keep_columns = self.columns  # 则返回所有列

    if additional_columns is None:
        additional_columns = {}

    # 结果的新表
    result_table = Table(keep_columns + additional_columns.keys())

    for row in self.rows:
        new_row = [row[column] for column in keep_columns]
        for column_name, calculation in additional_columns.iteritems():
            new_row.append(calculation(row))
        result_table.insert(new_row)

    return result_table
```

select() 会返回新的表，而一般的 SQL SELECT 仅会生成某种临时结果集（除非你显式地将结果导入一个表）。

我们也需要 where() 和 limit() 方法。两种方法都很简单：

```
def where(self, predicate=lambda row: True):
    """return only the rows that satisfy the supplied predicate"""
    where_table = Table(self.columns)
    where_table.rows = filter(predicate, self.rows)
    return where_table

def limit(self, num_rows):
    """return only the first num_rows rows"""
    limit_table = Table(self.columns)
    limit_table.rows = self.rows[:num_rows]
    return limit_table
```

然后我们可以轻松创建与先前的 SQL 语句等价的 NotQuiteABase 语句：

```
# SELECT * FROM users;
users.select()

# SELECT * FROM users LIMIT 2;
users.limit(2)

# SELECT user_id FROM users;
users.select(keep_columns=["user_id"])

# SELECT user_id FROM users WHERE name = 'Dunn';
users.where(lambda row: row["name"] == "Dunn") \
```

```
        .select(keep_columns=["user_id"])

# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row): return len(row["name"])

users.select(keep_columns=[],
             additional_columns = { "name_length" : name_length })
```

注意——不同于本书其他部分——我在这里用反斜线 \ 来表示程序语句的跨行延续。与其他方法相比，我认为这种方法可以让串连在一起的 NotQuiteABase 查询语句更易读。

23.5 GROUP BY

另一种常见的 SQL 操作是 GROUP BY，它可以将在特定列有相同值的行进行分组，并求出特定的汇总值，如 MIN、MAX、COUNT 或 SUM。（回想 10.3 节“处理数据”中提到的 group_by 函数。）

例如，你需要对每个可能的名字长度找出相应的用户数目和最小 user_id：

```
SELECT LENGTH(name) as name_length,
 MIN(user_id) AS min_user_id,
 COUNT(*) AS num_users
FROM users
GROUP BY LENGTH(name);
```

我们通过 SELECT 生成的每个字段，要么需要在 GROUP BY 语句中完成（name_length 就是这样），要么需要汇总计算（min_user_id 和 num_users 就是这样）。

SQL 同样支持 HAVING 子句，它和 WHERE 子句类似，只是前者只对汇总结果过滤（而后者在汇总计算之前就过滤）。

也许你想知道名字以某些特定字母开头的用户的平均朋友数，结果却只看到了平均个数大于 1 的结果。（是的，其中某些例子是人为的。）

```
SELECT SUBSTR(name, 1, 1) AS first_letter,
 AVG(num_friends) AS avg_num_friends
FROM users
GROUP BY SUBSTR(name, 1, 1)
HAVING AVG(num_friends) > 1;
```

（处理字符串的函数会基于不同的 SQL 实现而有所不同；一些数据库会用 SUBSTRING 或者别的方式。）

你也可以计算总体的汇总值，这时我们不用 GROUP BY：

```
SELECT SUM(user_id) as user_id_sum
FROM users
WHERE user_id > 1;
```

为了对 NotQuiteABase 表增加这项函数功能，我们增加一个 `group_by()` 方法。它首先取你希望分组的列名、你希望对每组运行的汇总函数的字典和作用于多行的可选判定函数 `having`。

然后完成以下步骤。

1. 生成默认字典，将（按值分组的）元组映射到行（包含按值分的组）。前面提过，列表不可以用作字典的键；你得使用元组。
2. 遍历表的每一行，填充 `defaultdict`。
3. 用正确的输出列生成新表。
4. 遍历 `defaultdict`，并填充输出表。使用 `having` 过滤（如果有的话）。

（实际的数据库会用更有效的方式完成这些步骤。）

```
def group_by(self, group_by_columns, aggregates, having=None):

    grouped_rows = defaultdict(list)

    # 填充组
    for row in self.rows:
        key = tuple(row[column] for column in group_by_columns)
        grouped_rows[key].append(row)

    # 结果表中包含组列与汇总
    result_table = Table(group_by_columns + aggregates.keys())

    for key, rows in grouped_rows.iteritems():
        if having is None or having(rows):
            new_row = list(key)
            for aggregate_name, aggregate_fn in aggregates.iteritems():
                new_row.append(aggregate_fn(rows))
            result_table.insert(new_row)

    return result_table
```

我们再来看看为了完成先前 SQL 语句的功能还能用什么别的方法。`name_length` 标准表示如下所示：

```
def min_user_id(rows): return min(row["user_id"] for row in rows)
stats_by_length = users \
    .select(additional_columns={"name_length" : name_length}) \
    .group_by(group_by_columns=["name_length"],
              aggregates={ "min_user_id" : min_user_id,
                           "num_users" : len })
```

`first_letter` 的标准表示是：

```
def first_letter_of_name(row):
    return row["name"][0] if row["name"] else ""
```

```
def average_num_friends(rows):
    return sum(row["num_friends"] for row in rows) / len(rows)

def enough_friends(rows):
    return average_num_friends(rows) > 1

avg_friends_by_letter = users \
    .select(additional_columns={'first_letter' : first_letter_of_name}) \
    .group_by(group_by_columns=['first_letter'],
              aggregates={ "avg_num_friends" : average_num_friends },
              having=enough_friends)
```

user_id_sum 的标准表示是：

```
def sum_user_ids(rows): return sum(row["user_id"] for row in rows)

user_id_sum = users \
    .where(lambda row: row["user_id"] > 1) \
    .group_by(group_by_columns=[],
              aggregates={ "user_id_sum" : sum_user_ids })
```

23.6 ORDER BY

你会经常需要对结果排序。比如，你也许想知道你前两个用户的名字（按字母顺序排序的）：

```
SELECT * FROM users
ORDER BY name
LIMIT 2;
```

可以通过给表提供一个具有排序（order）功能的 order_by() 方法来轻易实现：

```
def order_by(self, order):
    new_table = self.select()       # 进行一次复制
    new_table.rows.sort(key=order)
    return new_table
```

然后我们可以这样做：

```
friendliest_letters = avg_friends_by_letter \
    .order_by(lambda row: -row["avg_num_friends"]) \
    .limit(4)
```

SQL 中的 ORDER BY 可以让你为每个字段设定 ASC（升序排列）或 DESC（降序排列）；这里我们不得不把它并入到 order 函数中。

23.7 JOIN

关系型数据库的表通常是正则化的，意味着依照冗余最小化的原则进行组织。例如，当我

们处理用户对 Python 的兴趣时，我们只能为每个用户提供一个包含他的兴趣的列表。

SQL 表并不会包含真正的列表，它的实现机制是生成第二个表 `user_interests`，这个表包含了从 `user_id` 到兴趣的一对多的关系。在 SQL 中，你可以这样做：

```
CREATE TABLE user_interests (
    user_id INT NOT NULL,
    interest VARCHAR(100) NOT NULL
);
```

而在 NotQuiteABase 中，你需要建立表：

```
user_interests = Table(["user_id", "interest"])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])
```

> 这样仍然有很多的冗余性——兴趣“SQL”存储在两个不同的地方。在实际数据库中，你需要将 `user\_id` 和 `interest\_id` 存在表 `user\_interests` 中，并建立将 `interest\_id` 映射到 `interest` 的第 3 个表 `interests`，这样你就可以将每个兴趣名字仅仅存储一次。这里，我们会举比实际需要更复杂的例子。

当我们的数据跨表存储时，该如果分析？当然是把表 `JOIN` 在一起。`JOIN` 可以将左表的行和右表对应的行结合在一起，其中，如何“对应”取决于我们对 join 的具体设定。

例如，为了找出对 SQL 感兴趣的用户，你需要：

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

JOIN 意味着对 `users` 中的每行都要找到 `user_id`，并找出 `user_interests` 中包含相同 `user_id` 的每一行，把它们联系起来。

注意，我们得指定哪个表需要参与 `JOIN`，哪些列需要被 join `ON`。这是一种 `INNER JOIN`，返回的是条件匹配行（仅仅是行的组合）的组合。

`LEFT JOIN` 不仅返回匹配行的组合，也返回左表中无匹配行的行（这种情形下，来自右表的字段值全部为 `NULL`）。

通过 `LEFT JOIN` 可以很容易计算出每个用户的兴趣数目：

```
SELECT users.id, COUNT(user_interests.interest) AS num_interests
FROM users
LEFT JOIN user_interests
ON users.user_id = user_interests.user_id
```

LEFT JOIN 保证没任何兴趣的用户在并集结果数据集中仍有一席之地（来自表 user_interest 的字段值为 NULL），并且 COUNT 只对非 NULL 值进行计数。

NotQuiteABase 的 join() 实现更为严格一些——它只对两表有共同列的部分做合并。虽然如此，也不妨把它写出来：

```
def join(self, other_table, left_join=False):

    join_on_columns = [c for c in self.columns           # 两个表的列
                       if c in other_table.columns]

    additional_columns = [c for c in other_table.columns  # 右表中的列
                          if c not in join_on_columns]

    # 左表中所有列 + 右表增加的列
    join_table = Table(self.columns + additional_columns)

    for row in self.rows:
        def is_join(other_row):
            return all(other_row[c] == row[c] for c in join_on_columns)

        other_rows = other_table.where(is_join).rows

        # 每对匹配的行生成一个新行
        for other_row in other_rows:
            join_table.insert([row[c] for c in self.columns] +
                              [other_row[c] for c in additional_columns])

        # 如果没有行匹配,在左并集的操作下生成空值
        if left_join and not other_rows:
            join_table.insert([row[c] for c in self.columns] +
                              [None for c in additional_columns])

    return join_table
```

这样我们就找到了对 SQL 感兴趣的用户：

```
sql_users = users \
    .join(user_interests) \
    .where(lambda row: row["interest"] == "SQL") \
    .select(keep_columns=["name"])
```

并且可以获得兴趣的数目：

```
def count_interests(rows):
    """counts how many rows have non-None interests"""
    return len([row for row in rows if row["interest"] is not None])
```

```
user_interest_counts = users \
    .join(user_interests, left_join=True) \
    .group_by(group_by_columns=["user_id"],
              aggregates={"num_interests" : count_interests })
```

在 SQL 中，还有一个 RIGHT JOIN，它查询的是右表中没有匹配的行。还有 FULL OUTER JOIN，它查询的是两表中无匹配的所有行。我们不再一一展示。

23.8 子查询

在 SQL 中，你可以把查询结果当成表，执行 SELECT（或 JOIN）操作。因此，如果你希望找到对 SQL 感兴趣的用户中的最小 user_id，可以考虑使用子查询。（当然，你也可以通过 JOIN 来做相同的运算，只是这样就无法展示子查询了。）

```
SELECT MIN(user_id) AS min_user_id FROM
(SELECT user_id FROM user_interests WHERE interest = 'SQL') sql_interests;
```

如果设计好了 NotQuiteABase，我们就可以方便地得到如下结果。（我们的查询结果是实际的表。）

```
likes_sql_user_ids = user_interests \
    .where(lambda row: row["interest"] == "SQL") \
    .select(keep_columns=['user_id'])

likes_sql_user_ids.group_by(group_by_columns=[],
                            aggregates={ "min_user_id" : min_user_id })
```

23.9 索引

为了找出包含特定值（比如名字是“Hero”）的行，NotQuiteABase 需要检查表的每一行。如果表有很多行，得花费很长时间。

同样，我们的 join 算法也极端低效。为了确认一个匹配，对左表的每一行，都需要检查右表的每一行。如果对象是两个大规模的表，这样的计算几乎没有结束的时候。

并且，你有时会需要对某些列加以约束。比如，在表 user 中，你很可能不希望两个不同的用户共用一个 user_id。

索引可以解决以上问题。如果表 user_interests 有一个基于 user_id 的索引，那么使用一个智能的 join 算法就可以不用遍历全表而直接完成匹配。如果表 users 已有基于 user_id 的“唯一”索引，你再插入重复记录时会报错。

数据库中的每个表都有一个或多个索引，这让你可以通过关键词列快速查找行，可以有效并表，以及可以对行或者行集合施加唯一约束。

设计好、用好索引从某种程度上来讲更像魔术（还会因具体数据库的不同而情况各异），但如果你需要处理大量数据库工作，学习一下索引还是值得的。

23.10 查询优化

回想一下找出所有对 SQL 感兴趣的用户的查询方法：

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

在 NotQuiteABase 中，有（至少）两种不同的方法可以写这个查询。你可以在并表之前过滤表 `user_interests`：

```
user_interests \
    .where(lambda row: row["interest"] == "SQL") \
    .join(users) \
    .select(["name"])
```

或者你可以对并表的结果过滤：

```
user_interests \
    .join(users) \
    .where(lambda row: row["interest"] == "SQL") \
    .select(["name"])
```

两种方法都可以获得相同的结果，但先过滤再并表的方式效率更高，因为这样可以在并表时操作更少的行。

在 SQL 中，你基本不用担心这个。你只需“声明”自己需要的结果，再把任务丢给查询引擎（和有效使用索引）来实现即可。

23.11 NoSQL

数据库的一个近期发展趋势是非关系型的“NoSQL”数据库，这种数据库不将数据存放在表中。例如 MongoDB 这种流行的无结构数据库，它的元素直接是一些复杂 JSON 文档，而不是行。

此外，还有以列而非行的形式存储数据的列型数据库（适用于数据本身有很多列但查询却很少用到的情形），优化的通过键来检索单独（复杂）的值的键值存储对数据库，用来存储和遍历图像的数据库，跨多个数据中心运行的优化数据库，在内存中运行的数据库，以及存储时间序列数据的数据库等上百个数据库。

未来瞬息万变，谁也无法预知明天会流行什么，所以我仅仅告诉你有 NoSQL 这么一回事。因此你现在知道有 NoSQL 这个东西就行了。

23.12　延伸学习

- 如果你想下载关系型数据库的相关练习，SQLite（http://www.sqlite.org/）是个很好的选择，它快捷而精致。MySQL（http://www.mysql.com/）和 PostgreSQL（http://www.postgresql.org/）则规模更大，功能更多。这些软件都是免费的，而且有大量文档可供参考。
- 如果你想深入学习 NoSQL，我推荐 MongoDB（https://www.mongodb.org/），它上手特别容易，不过这个特点既受赞扬又被诟病。这个软件也拥有非常友好的文档。
- 维基百科上关于 NoSQL 的文章（https://en.wikipedia.org/wiki/NoSQL）非常全面，现在其所提供的链接涉及的有些数据库在撰写本书时甚至还没诞生。

第24章

MapReduce

明天已经照耀现在，只是尚未洒满每个角落。

——威廉·吉布森

MapReduce 是一个用来在在大型数据集上执行并行处理的算法模型。尽管这是一个非常强大的技术，但它的基本原理却很简单。

假设我们有一组待处理的项目，这些项目可能是网页日志、许多本书的文本、图像文件或者是其他东西。一个基本的 MapReduce 算法包括下面几个步骤。

1. 使用 `mapper` 函数把每个项目转化成零个或多个键值对。（通常这被称为 `map` 函数，但是已经有了一个叫 `map` 的 Python 函数，所以我们不能把它们混淆。）
2. 用相同的键把所有的键值对收集起来。
3. 在每一个分好组的值集合上使用 `reducer` 函数，对每个对应的键生成输出值。

这样讲是很抽象的，所以让我们看一个具体的例子。数据科学里很少有绝对的规则，但有一个不成文的规则是，你面对的第一个 MapReduce 的例子中必须要涉及单词计数。

24.1　案例：单词计数

DataSciencester 网站的用户数量已增长到了数百万！这对你的工作是极大的保障，但也使日常分析变得有点困难了。

比如，内容部门的副总想知道用户的状态更新都涉及什么内容。作为初步的尝试，你决定

统计一下出现的单词的数量，这样你就可以准备一个关于出现频度最高的单词的报告。

当有几百个用户的时候这做起来很简单：

```
def word_count_old(documents):
    """word count not using MapReduce"""
    return Counter(word
        for document in documents
        for word in tokenize(document))
```

而数百万个用户的 documents 集合就变得很大，你的电脑都装不下。如果你能把它们纳入 MapReduce 模型处理，就可以使用你的引擎上已经部署的一些“大数据”架构。

首先，我们需要一个函数来把文档转化成一系列的键值对。我们希望输出的结果能按单词分组，这意味着键应该是单词。对每一个单词，我们只发送值 1 来表示这个键值对对应于单词出现一次：

```
def wc_mapper(document):
    """for each word in the document, emit (word,1)"""
    for word in tokenize(document):
        yield (word, 1)
```

先暂时跳过第二步，假设对某些词我们已经收集了所发送过的对应计数的列表。接下来生成我们所需要的这个词的全部计数：

```
def wc_reducer(word, counts):
    """sum up the counts for a word"""
    yield (word, sum(counts))
```

再返回步骤 2，现在我们需要收集来自 wc_mapper 的结果，再把它们传递给 wc_reducer。让我们思考一下怎么在一台计算机上完成这件事：

```
def word_count(documents):
    """count the words in the input documents using MapReduce"""

    # 存放分好组的值
    collector = defaultdict(list)

    for document in documents:
        for word, count in wc_mapper(document):
            collector[word].append(count)

    return [output
            for word, counts in collector.iteritems()
            for output in wc_reducer(word, counts)]
```

假设我们有三个文档 ["data science", "big data", "science fiction"]。

然后把 wc_mapper 应用到第一个文档，产生两个键值对 ("data", 1) 和 ("science", 1)。在

处理完所有三个文档之后，列表 collector 会包含：

```
{ "data" : [1, 1],
  "science" : [1, 1],
  "big" : [1],
  "fiction" : [1] }
```

然后 wc_reducer 函数生成了每个单词的计数：

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

24.2　为什么是MapReduce

如同早先所提到的，MapReduce 的主要优点就是通过将处理过程移动到数据来进行分布式的计算。假设我们想对数以十亿计的文档进行单词计数。

我们最初的（非 MapReduce 的）方法要求机器在每一个文档上进行处理。这意味着所有的文档要么存在机器上要么在处理期间转移到机器上。更重要的是，这意味着机器一次只能处理一个文档。

> 如果机器是多核的，且如果代码是为了利用多核的优势而重写过的，那它是有可能一次处理几个文档的。但即使这样，所有的文档也都要放到机器中来。

假设现在我们将数十亿个文档分散到 100 台机器上。利用正确的架构（并且掩盖掉一些细节），我们可以做下面的事。

- 让每一台机器在它的文档上运行 mapper 函数，产生大量的键值对。
- 把那些键值对分配到一些“reducing”的机器上，确保对应任何一个给定键的对在同一台机器上完成计算。
- 每一台 reducing 机器通过键分组这些对，然后对每个值的集合运行 reducer 函数。
- 返回每个键值对。

它可以处理的横向规模水平令人惊叹。如果我们把机器的数量加倍，那么（忽略运行 MapReduce 系统的某些固定成本）计算的运行速度大约会快两倍。每一个 mapper 机器只需要做一半的工作，（假设有足够多不同的键来进一步分配 reducer 工作）reducer 机器也是。

24.3　更加一般化的MapReduce

思考一下，前面例子中所有的单词计数代码是包括在 wc_mapper 和 wc_reducer 这两个函数中的。这意味着通过几个改变我们会得到一个更通用的框架（仍然是运行在单个机器

上的）：

```
def map_reduce(inputs, mapper, reducer):
    """runs MapReduce on the inputs using mapper and reducer"""
    collector = defaultdict(list)

    for input in inputs:
        for key, value in mapper(input):
            collector[key].append(value)

    return [output
            for key, values in collector.iteritems()
            for output in reducer(key,values)]
```

然后我们可以通过下面的方法简单地完成单词计数：

```
word_counts = map_reduce(documents, wc_mapper, wc_reducer)
```

这为我们解决各种类型的问题提供了灵活性。

在继续讲解下面的内容之前，我们先观察一下 wc_reducer 函数，它仅仅是把对应于每一个键的值加起来。这种聚合是非常普遍的，值得把它抽象出来：

```
def reduce_values_using(aggregation_fn, key, values):
    """reduces a key-values pair by applying aggregation_fn to the values"""
    yield (key, aggregation_fn(values))

def values_reducer(aggregation_fn):
    """turns a function (values -> output) into a reducer
    that maps (key, values) -> (key, output)"""
    return partial(reduce_values_using, aggregation_fn)
```

在这之后我们就可以轻松创建以下内容：

```
sum_reducer = values_reducer(sum)
max_reducer = values_reducer(max)
min_reducer = values_reducer(min)
count_distinct_reducer = values_reducer(lambda values: len(set(values)))
```

24.4 案例：分析状态更新

内容部门的副总对单词计数印象深刻，并询问你还能从用户的状态更新中学到什么。你设法提取了一个类似下面这样的状态更新的数据集：

```
{"id": 1,
 "username" : "joelgrus",
 "text" : "Is anyone interested in a data science book?",
 "created_at" : datetime.datetime(2013, 12, 21, 11, 47, 0),
 "liked_by" : ["data_guy", "data_gal", "mike"] }
```

假设我们想找出每周内的哪一天人们讨论数据科学最多。为了找到这个结果，只需计算一下一周内每一天有多少个数据科学更新。这意味着我们需要按每周内的每天进行分组，这就是我们的键。而且如果对每一个包含“数据科学”的更新发送一个值 1，就可以使用 sum 函数得到总数：

```
def data_science_day_mapper(status_update):
    """yields (day_of_week, 1) if status_update contains  "data  science" """
    if "data science" in status_update["text"].lower():
        day_of_week = status_update["created_at"].weekday()
        yield (day_of_week, 1)

data_science_days = map_reduce(status_updates,
                               data_science_day_mapper,
                               sum_reducer)
```

再举一个稍微复杂一点的例子，假设我们需要找出每个用户在其状态更新中最常用的单词是什么。为了建立 mapper，我们的脑海中会浮现以下三种方法。

- 把用户名放到键当中；把单词和计数放到值当中。
- 把单词放到键当中；把用户名和计数放到值当中。
- 把用户名和单词放到键当中；把计数放到值当中。

稍作考虑，我们很容易就能判断出应该按 username 分组，因为我们想分别考虑每个人的单词。而且我们不想用 word 来分组，因为我们的 reducer 需要看到每个人所有的词汇，以此找到哪一个是最流行的。这意味着第一种选择是最优选择：

```
def words_per_user_mapper(status_update):
    user = status_update["username"]
    for word in tokenize(status_update["text"]):
        yield (user, (word, 1))

def most_popular_word_reducer(user, words_and_counts):
    """given a sequence of (word, count) pairs,
    return the word with the highest total count"""

    word_counts = Counter()
    for word, count in words_and_counts:
        word_counts[word] += count

    word, count = word_counts.most_common(1)[0]

    yield (user, (word, count))

user_words = map_reduce(status_updates,
                        words_per_user_mapper,
                        most_popular_word_reducer)
```

或者我们能为每个用户找到各自的状态点赞者的个数：

```
def liker_mapper(status_update):
    user = status_update["username"]
    for liker in status_update["liked_by"]:
        yield (user, liker)

distinct_likers_per_user = map_reduce(status_updates,
                                      liker_mapper,
                                      count_distinct_reducer)
```

24.5 案例：矩阵计算

回想 22.2.1 节“矩阵乘法”，给定一个 $m \times n$ 的矩阵 A 和一个 $n \times k$ 的矩阵 B，可以把它们乘起来得到 $m \times k$ 的矩阵 C，其中 C 的第 i 行第 j 列的元素由下式给出：

$$C_{ij}=A_{i1}B_{1j}+A_{i2}B_{2j}+\cdots+A_{in}B_{nj}$$

如同我们之前所见的，表示一个 $m \times n$ 矩阵的“自然的”方法是列表的列表，其中元素 A_{ij} 是第 i 个列表的第 j 个元素。

但大型矩阵有时候是稀疏的，即大部分的元素等于 0。对于大型稀疏矩阵而言，列表的列表是一种非常浪费的表达方式。一种更简洁的表达方式是元组的列表 (`name`，`i`，`j`，`value`)，其中 `name` 代表矩阵，而 `i`、`j`、`value` 表示一个非零元素的位置。

比如，一个十亿 × 十亿的矩阵会有亿亿级别（quintillion，1×10^{18}）的元素，这是难以存储在一个计算机中的。但是如果每行当中只有不多的一些非零元素，上面那种替代的表示法就会小很多个数量级。

基于这种表示法，我们可以使用 MapReduce 以分布式的方式执行矩阵乘法。

为使用这种算法，请注意，A_{ij} 只用于计算 C 的第 i 行的元素，B_{ij} 只用于计算 C 的第 j 列的元素。我们的目标是使 `reducer` 的每一个输出构成矩阵 C 的一个元素。这意味着我们需要用 `mapper` 发送键值，以确定 C 中的每个元素。建议像下面这样处理：

```
def matrix_multiply_mapper(m, element):
    """m is the common dimension (columns of A, rows of B)
    element is a tuple (matrix_name, i, j, value)"""
    name, i, j, value = element

    if name == "A":
        # A_ij是每个C_ik之和的第j个元素,其中k=1..m
        for k in range(m):
            # 与C_ik的其他元素分组
            yield((i, k), (j, value))
    else:
        # B_ij是每个C_kj之和的第i个元素
        for k in range(m):
            # 与C_kj的其他元素分组
            yield((k, j), (i, value))
```

```
def matrix_multiply_reducer(m, key, indexed_values):
    results_by_index = defaultdict(list)
    for index, value in indexed_values:
        results_by_index[index].append(value)

    # 对有两个结果的位置把所有的乘积加起来
    sum_product = sum(results[0] * results[1]
                      for results in results_by_index.values()
                      if len(results) == 2)

    if sum_product != 0.0:
        yield (key, sum_product)
```

比如，如果你有如下的两个矩阵：

```
A = [[3, 2, 0],
     [0, 0, 0]]

B = [[4, -1, 0],
     [10, 0, 0],
     [0, 0, 0]]
```

你可以把它们重写为元组：

```
entries = [("A", 0, 0, 3), ("A", 0, 1,  2),
           ("B", 0, 0, 4), ("B", 0, 1, -1), ("B", 1, 0, 10)]
mapper = partial(matrix_multiply_mapper, 3)
reducer = partial(matrix_multiply_reducer, 3)

map_reduce(entries, mapper, reducer) # [((0, 1), -3), ((0, 0), 32)]
```

在这样一个小矩阵上操作并不太有趣，但是如果你有百万行百万列的矩阵，MapReduce 就会起很大作用。

24.6 题外话：组合器

你很可能已经注意到，许多 mapper 可能包括一些额外的信息。比如，在计数单词的时候，与其发送 `(word, 1)` 并累加求和，不如发送 `(word, None)` 再取其长度。

我们没有这么做的一个原因是，在分布式情景下，我们有时会想用组合器（combiner）来缩减在计算机之间转移的数据数量。如果一个 mapper 机器发现单词“data”500 次，可以让它在移交数据给缩减机器之前把 500 个 `("data", 1)` 组合成一个单独的 `("data", 500)`。这使得转移的数据少了很多，算法会大大加快。

基于我们编写 reducer 的方法，它能正确地处理这些组合的数据。（如果我们已经用 `len` 函数写了 reducer，它就不能处理组合了。）

24.7　延伸学习

- 使用最为广泛的 MapReduce 系统是 Hadoop（http://hadoop.apache.org/），它值得用很多本书来阐述。它有许多商业或非商业的发行版，以及一个由 Hadoop 相关工具组成的巨大的生态系统。

  为了使用 Hadoop，你需要建立自己的聚类（或者可以在允许的情况下使用别人建好的聚类），当然，对于承压能力较差的人来说，可以不把这当成必然的任务。Hadoop mapper 和 reducer 通常是用 Java 写成的，尽管有一种被称为“Hadoop streaming”的功能允许你用其他的语言（包括 Python）来写。
- Amazon.com 提供 Elastic MapReduce（http://aws.amazon.com/cn/elasticmapreduce/）服务，它可以用编程的方式来创建或销毁聚类，并只根据你使用该服务的时长来收费。
- mrjob（https://github.com/Yelp/mrjob）是 Python 的一个 Hadoop（或 Elastic MapReduce）的接口包。
- Hadoop 任务是典型的高延迟的，这对于“实时分析”来说不是个好选择。有多种建立在 Hadoop 之上的“实时分析”工具，同时还有一些替代性的框架也日益流行。最流行的两种是 Spark（http://spark.apache.org/）和 Storm（http://storm.incubator.apache.org/）。
- 总之，很有可能目前流行的某种新的分布式框架在本书写作时还未问世，那就需要你自己把它找出来了。

第 25 章

数据科学前瞻

此刻，我再次祈求我丑陋的后代生生世世繁荣昌盛。

——玛丽·雪莱

从这里出发，要去哪里呢？如果至此讲的有关数据科学的东西还没有把你吓跑的话，接下来你可以学习以下内容。

25.1 IPython

在本书前面的章节我们提到过 IPython（http://ipython.org/）。它提供了一个远比标准 Python shell 功能强大的 shell，而且加入了一些“魔法函数”，可以让你（比如）轻松地复制粘贴代码（以空行和空白的组合形式实现代码通常是比较复杂的），而且可以在 shell 内部运行代码。

掌握了 IPython 会让你的工作变得非常轻松。（甚至仅仅学一点点 IPython 就可以了。）

此外，它还允许你创建将文本、Python 动态代码和可视化相结合的“日记本”（notebook），你可以将它们与别人共享，或仅仅保存为自己的日志（图 25-1）。

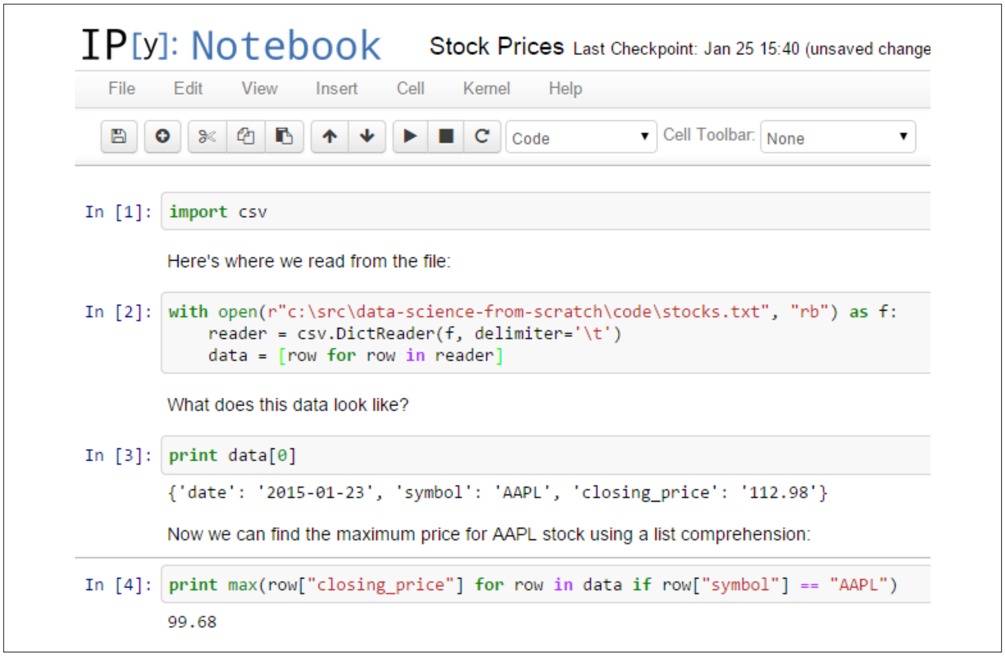


图 25-1：一个 IPython 日记本

25.2　数学

本书涵盖了线性代数（第 4 章）、统计（第 5 章）、概率（第 6 章）和机器学习的一些内容。

要想成为一名优秀的科学家，你需要知道更多关于这些领域的知识，而且我鼓励你对每一个领域开展深入的学习。你可以参考我在每一章末尾推荐的教科书，也可以使用自己选择的教科书，或通过在线课程甚至线下课程来进行学习。

25.3　不从零开始

“从零开始”做一件事情对于理解这件事情的工作原理是很有好处的。但这种方法对于性能表现（除非你仅仅是出于性能方面的考虑而实现它们）、易用性、快速成型或误差处理来说并不是很理想。

在实践中，你可能需要用到精心设计的能稳定地实施基础原则的库。（我本来是想在本书中加一个“让我们来学一些库”的板块，幸好，被 O’Reilly 否决了。）

25.3.1　NumPy

NumPy（即“Numeric Python”，http://www.numpy.org/）提供了处理“真实”科学计算的

工具。它提供了比我们的 `list` 向量性能更好的数组，提供了比我们的 `list-of-list` 矩阵性能更好的矩阵，以及大量利用它们来工作的数值函数。

对很多其他的库来说，NumPy 是个基础构件，这是它特别值得学习的原因。

25.3.2 pandas

pandas（http://pandas.pydata.org/）提供了处理 Python 数据集的更多的数据结构。它主要的抽象概念是 DataFrame，在内容上与我们在第 23 章中构建的 NotQuiteABase `table` 类很相似，但是有更多的功能和更好的性能。如果你打算用 Python 修改、分划、分组或操作数据集，pandas 是一个非常有用的工具。

25.3.3 scikit-learn

scikit-learn（http://scikit-learn.org/）可能是 Python 中处理机器学习问题最常用的库。它包括我们用过的所有模型以及很多我们没用过的模型。在真实的问题上，你无需从零开始建立决策树，而是可以使用 scikit-learn 来做繁重的工作。在真实的问题上，你也无需手动写出优化算法，而是可以依靠 scikit-learn 使用已有的优秀算法。

scikit-learn 的文档包括了许许多多案例（http://scikit-learn.org/stable/auto_examples/）来说明它可以做什么（或者，更一般地，说明机器学习可以做什么）。

25.3.4 可视化

我们创建过的 `matplotlib` 图形很清晰而且功能强大，但它还不够美观（而且一点交互性也没有）。如果想深入了解数据可视化，你可以有多种选择。

第一种是深入学习 `matplotlib`，因为我们涉及的内容只是它的一小部分。在它的网站上有许多关于它的功能的例子（http://matplotlib.org/examples/），还有一些更有趣的可视化的图库（http://matplotlib.org/gallery.html）。如果你打算创建静态的可视化（比如想把它印在书里），这可能是你下一步最好的选择。

你也应该试试 seaborn（http://web.stanford.edu/~mwaskom/software/seaborn/），这是一个可以使 `matplotlib` 更有吸引力（还有其他很多优点）的库。

如果你想创建那种可以在网络上分享的*交互式可视化*，D3.js（http://d3js.org/）是首选，它是一个可用于创建“数据驱动文档”（data driven documents，名称中的“3D”便由此得来）的 JavaScript 库。即使你不太懂 JavaScript，通常也可以从 D3 的图库（https://github.com/mbostock/d3/wiki/Gallery）中找到可以模仿的例子，并套用到你的数据上。（好的数据科学家从 D3 的图库中复制例子，出色的数据科学家从 D3 的图库中“*偷*”例子。）

即使你对 D3 一点兴趣也没有，仅仅浏览一下它的图库也能学到不少关于数据可视化的东西。

Bokeh（http://bokeh.pydata.org/en/latest/）是一个把 D3 风格的功能整合到 Python 中的项目。

25.3.5　R

尽管你可以完全不用学习 R（http://www.r-project.org/），但许多数据科学家和数据科学项目都会用到 R，所以起码应该熟悉它。

这一部分是因为这样做有益于你理解别人基于 R 的博客、案例和代码，一部分是因为这样可以让你更好地领略 Python 的（相对的）清晰和优雅。还有一部分原因是，这会让你在永不停息的“R 好还是 Python 好”的口水战中成为一个更加见多识广的专家。

这个世界从不缺乏 R 的教程、R 的课程和 R 的图书。我听说 *Hands-On Programming with R*（http://shop.oreilly.com/product/0636920028574.do）这本书不错，不仅仅是因为它也是 O'Reilly 出版的书。（好吧，主要就是因为它是 O’Reilly 出版的书。）

25.4　寻找数据

如果你把从事数据科学作为你工作的一部分，那么你会很愿意把获得数据也作为你工作的一部分（尽管并不一定）。如果你把从事数据科学工作视为乐趣将会如何？数据是无处不在的，但下面这些资源可以是很好的起点。

- Data.gov（http://www.data.gov/）是政府开放数据的门户网站。如果你想找任何和政府有关的数据（现在看这可能涉及方方面面的事情），它是个很好的开始。
- reddit 上有 r/datasets（https://www.reddit.com/r/datasets）和 r/data（http://www.reddit.com/r/data）两个论坛，是一个可以请求数据和发现数据的地方。
- Amazon.com 上有一些公用数据集（http://aws.amazon.com/cn/public-data-sets/），它希望你能使用它的产品来分析这些数据集（但你可以用任何你想用的产品来分析）。
- Robb Seaton 的博客上有一个富有创意的专业数据集的列表（http://rs.io/100-interesting-data-sets-for-statistics/）。
- Kaggle（https://www.kaggle.com/）是一个举办数据科学竞赛的网站。我从没成功跻身这个比赛（在数据科学领域我没有多少竞争力），但没准你就会成功。

25.5　从事数据科学

翻阅数据目录固然不错，但是最好的项目（和产品）还是那些让人心动的。下面列举我做过的其中一些项目。

25.5.1　Hacker News

Hacker News（https://news.ycombinator.com/news）是一个聚集并讨论与技术相关的新闻的网站。它收集了大量文章，但很多对我来说并不有趣。

因此，数年之前，我着手建立了一个 Hacker News 的内容分类器（https://github.com/joelgrus/hackernews）来预测我是否喜欢某篇给定的文章。这一做法可能会受到有些 Hacker News 用户的排斥，他们会抱怨怎么还会有人不喜欢 Hacker News 的所有文章。

这项工作涉及对大量文章的手动标注（为了建立训练集），选择文章的特征（比如标题中的单词和链接的域），以及训练一个和之前我们做过的垃圾邮件检测类似的朴素贝叶斯分类器。

不知道出于什么心理，我当时是用 Ruby 建立的分类器。吸取我的教训吧。

25.5.2　消防车

我生活在西雅图市中心的一条主要街道上，这条街是从一个消防站去往城市里大多数火灾（或者看起来疑似火灾）的必经之地。由此，多年来我产生了对西雅图消防部门的兴趣。

幸运的是（从数据的观点），消防部门运行着一个实时的 911 网站（http://www2.seattle.gov/fire/realtime911/getDatePubTab.asp），上面列出了每一次火警状况和出动的消防车。

所以，为了满足我的兴趣，我抓取了多年的火警数据并执行了关于消防车的社交网络分析（https://github.com/joelgrus/fire），这就要求我创造了一个特定的关于消防车的中心性的概念，我称之为 TruckRank。

25.5.3　T恤

我有一个小女儿，她的童年有一件令我感到十分沮丧的事情，就是“女孩的 T 恤”大都很单调乏味，而“男孩的 T 恤”却都充满趣味。

尤其是，我觉得出售给男童和女童的 T 恤之间有很明显的区别。所以我问自己能不能训练一个模型来识别这些区别。

剧透：我能（https://github.com/joelgrus/shirts）。

这项工作包括下载数百件 T 恤的图案，把它们改成同样的尺寸，再把它们转换为像素颜色的向量，最后使用逻辑回归建立分类器。

一种看起来较为简单的方法是针对每件 T 恤的颜色分类；第二种方法是找到 T 恤颜色向量

的前 10 个主成分，然后把每件 T 恤投影到由 10 个“特征 T 恤”（eigenshirt）[1] 所组成的 10 维空间上进行分类（见图 25-2）。



图 25-2：对应于第一个主成分的特征 T 恤

25.5.4 你呢？

什么事情会让你兴致勃发？什么问题会让你夜不能寐？去寻找相关的数据（或者抓取一些网站），对它们做一些数据科学的分析吧。

告诉我你的发现吧！通过 joelgrus@gmail.com 给我发邮件，或者去 Twitter @joelgrus 找我。

注 1：即 10 个主成分。——译者注

作者简介

Joel Grus 是 Google 的一位软件工程师，曾于数家创业公司担任数据科学家。目前住在西雅图，专注于数据科学工作并乐此不疲。偶尔在 joelgrus.com 发表博客，长期活跃于 Twitter @joelgrus。

关于封面

本书封面上的动物是岩雷鸟（学名 Lagopus muta），是松鸡家族中一种中等体型的猎鸟，在英国和加拿大叫“雷鸟”，在美国叫“雪鸡”。岩雷鸟喜定居，栖息于北极和靠近北极的欧亚大陆，以及北美的格陵兰岛地区，栖息地多为贫瘠而孤立之地，如苏格兰山脉、比利牛斯山、阿尔卑斯山、乌拉尔山脉、帕米尔高原、保加利亚、阿尔泰山脉和日本阿尔卑斯山脉。食物主要是桦树和柳树的芽，也包含种子、花、叶子、浆果等。发育中的雏鸟也吃昆虫。

雄性岩雷鸟并没有一般松鸡所具有的典型的羽饰，但它们有肉冠，可以帮助其求偶或与其他雄鸟争斗。大量研究证明，雄性岩雷鸟的肉冠尺寸与其睾酮水平之间有一定的相关性。岩雷鸟的羽毛冬季为白色，春夏会变为黑褐色，可起到季节性的保护作用。具有繁殖能力的雄鸟翅膀为白色，上半部分为灰色，但在冬季，除了尾羽呈黑色外，周身均为白色。

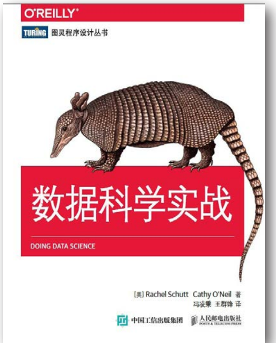
6 个月大的岩雷鸟即发育成熟，且通常每只雌鸟可孵化 6 只雏鸟，以帮助保护种群数量免受诸如狩猎等外界因素的影响。岩雷鸟主要被金雕捕食，而偏远孤立的栖息地帮助其躲过了很多其他捕食者。

岩雷鸟肉在冰岛是备受欢迎的节日餐主食。由于种群数量的下降，岩雷鸟的捕猎在 2003 年和 2004 年是被禁止的。2005 年，捕猎禁令解除，但特定时期内仍禁止捕猎。所有与岩雷鸟有关的交易都是非法的。

O’Reilly 封面上的动物很多都是濒临灭绝的，所有这些动物对世界来说都是很重要的。若想了解你力所能及的事，请访问 animals.oreilly.com。

封面图片来自 Cassell 的 *Natural History*。

延展阅读



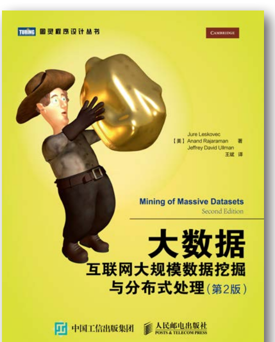
大数据时代的实战宝典，揭秘数据科学相关的最新算法、方法与模型！
本书脱胎于哥伦比亚大学“数据科学导论”课程的教学讲义，界定了数据科学的研究范畴，是一本多角度、全方位、深入介绍数据科学的实用指南。本书旨在让读者能举一反三地解决重要问题，内容包括数据科学及工作流程、统计模型与机器学习算法、信息提取与统计变量创建、数据可视化与社交网络、预测模型与因果分析、数据预处理与工程方法。

书号：978-7-115-38349-5
定价：79.00 元



大数据时代，数据科学研究与分析日益重要。本书独树一帜，教你利用灵活的命令行工具成为高效多产的数据科学家。为此，作者开发了数据科学工具箱，一个包含80多个命令行工具的安装简单的虚拟环境，能在Windows、OS X和Linux操作系统上运行。你将学会如何结合使用这些小而强大的命令行工具，快速地获取、清洗、探索和建模数据。即使你已经能够使用Python或R得心应手地处理数据，利用命令行也将大大改进你的数据科学工作流。

书号：978-7-115-39168-1
定价：49.00 元



大数据权威著作全面升级版！第1版畅销40000册！
本书源自作者在斯坦福大学教授的“海量数据挖掘”（CS246: Mining Massive Datasets）课程，第1版上市以来受到读者广泛欢迎和认可。本书以大数据环境下的数据挖掘和机器学习为重点，全面介绍了行之有效的数据处理算法，是在校学生和相关从业人员的必备读物。

书号：978-7-115-39525-2
定价：79.00 元



这是一本实用手册，四位作者均是Cloudera公司的数据科学家，他们联袂展示了利用Spark进行大规模数据分析的若干模式，而且每个模式都自成一体。首先介绍了Spark及其生态系统，接着详细介绍了将分类、协同过滤及异常检查等常用技术应用于基因学、安全和金融领域的若干模式。如果你对机器学习和统计学有基本的了解，并且会用Java、Python或Scala编程，这些模式将有助于你开发自己的数据应用。

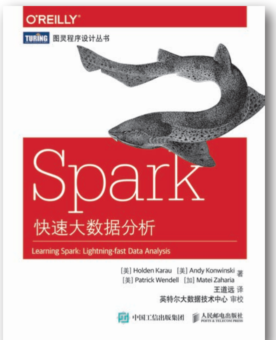
书号：978-7-115-40474-9
定价：59.00 元

延展阅读



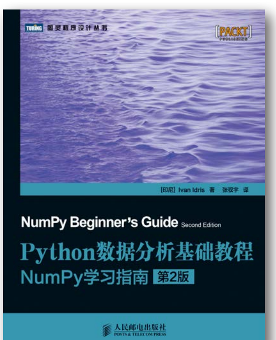
本书为了解数据可视化的重要内容和功能提供了多学科的视角，通过各种各样的案例分析，来演示可视化如何让数据变得更清晰、更全面，通过对数据可视化的广泛用途和适用性的讨论，来了解它如何让数据变得更容易接受和理解。
本书的读者对象包括数据分析师、视觉设计师，以及对数据呈现感兴趣的开发人员等。

书号：978-7-115-41470-0
定价：69.00 元



如今，所有领域的数据量都在急剧增长。针对如何高效利用这些数据，本书介绍了开源集群计算系统Apache Spark，它可以加速数据分析的实现和运行。利用Spark，你可以用Python、Java以及Scala的简易API来快速操控大规模数据集。本书由Spark开发者及核心成员共同打造，带领读者快速掌握用Spark收集、计算、简化和保存海量数据的方法，学会交互、迭代和增量式分析，解决分区、数据本地化和自定义序列化等问题。

书号：978-7-115-40309-4
定价：59.00 元



手把手带你进入大数据挖掘领域，Python数据分析首选！
NumPy是一个优秀的科学计算库，提供了很多实用的数学函数、强大的多维数组对象和优异的计算性能。本书从NumPy安装讲起，逐渐过渡到数组对象、常用函数、矩阵运算、线性代数、金融函数、窗函数、质量控制等内容，致力于向初中级Python编程人员全面讲述NumPy及其使用。

书号：978-7-115-33940-9
定价：49.00 元



为什么要将命运交予偶然之手？何不多学习一点概率知识，掌控自己的命运呢？本书会把你武装起来。
利用统计推断的各种工具，能够揭开概率的神秘面纱、发现相关性，能够异常准确地预测事件的发生，甚至能让你在博彩时准确下注，小有斩获。本书介绍的实用技巧运用了统计学原理，还借鉴了教育学和心理学上的测量和实验研究方法。这些技巧可以帮你解决商业、游戏以及日常生活中的各类问题。

书号：978-7-115-35621-5
定价：59.00 元

O'REILLY®

数据科学入门

数据科学是一个蓬勃发展、前途无限的行业，有人将数据科学家称为“21世纪头号性感职业”。本书从零开始讲解数据科学工作，教授数据科学工作所必需的黑客技能，并带领读者熟悉数据科学的核心知识——数学和统计学。

作者选择了功能强大、简单易学的Python语言环境，亲手搭建工具和实现算法，并精心挑选了注释良好、简洁易读的实现范例。书中涵盖的所有代码和数据都可以在GitHub上下载。

通过阅读本书，你可以：

- **学到一堂Python速成课；**
- **学习线性代数、统计和概率论的基本方法，了解它们是怎样应用在数据科学中的；**
- **掌握如何收集、探索、清理、转换和操作数据；**
- **深入理解机器学习的基础；**
- **运用*k*近邻、朴素贝叶斯、线性回归和逻辑回归、决策树、神经网络和聚类等各种数据模型；**
- **探索推荐系统、自然语言处理、网络分析、MapReduce和数据库。**

Joel Grus是Google的一位软件工程师，曾在多家创业公司担任数据科学家。目前住在西雅图，专注于数据科学工作并乐此不疲。偶尔在joelgrus.com发表博客，长期活跃于Twitter，可关注@joelgrus。

“Joel Grus带你踏上数据科学的探索之旅，让你从对数据的好奇飞跃到通透理解每个数据科学家都应知道的实用算法。”

——Rohit Sivaprasad
Soylent公司数据科学家
datatau.com

DATA/DATA SCIENCE

封面设计：Karen Montgomery **张健**

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 **计算机／数据分析**

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-41741-1

ISBN 978-7-115-41741-1
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks